Bleeding Edge Press

Building Apps with RIOT

Collin Green, Ryan Lee,
John Nolette, James Sparkman
& Joseph Szczesniak

Building Apps with Riot

By John Nolette, Collin Green, Ryan Lee, James Sparkman, and Joseph Szczesniak

PURCHASE THE FULL VERSION OF THE BOOK

If you enjoy this sample and would like to learn more about building apps with Riot, you can purchase the full version of the book at **Bleeding Edge Press**. We are offering a 10% discount with the code RIOTSITE.

Table of Contents

Understanding basic concepts 1

This chapter covers the basic aspects of the Riot framework, helping you build a foundation for following along with the rest of the book. We'll be covering the creation of a tag (component), a tag's lifecycle, mixins, and expressions. Many of the code samples provided in this chapter can be run directly within your browser using Riot's in-browser compiler. By the end of this chapter, you should feel comfortable enough to write a basic Riot tag, and have a clear understanding of Riot's tag ecosystem.

If you find yourself uneasy with seemingly more advanced topics, there is no need for panic. These topics are covered in detail throughout this book. This chapter is just an introduction to help familiarize yourself with what Riot has to offer.

You can access the code from our sample application included with this book at: (**https://github.com/backstopmedia/riot-book-example**).

Creating a tag

Tags are the core of the Riot framework, and are synonymous with web components. A tag allows you to encapsulate your HTML, styles, and JavaScript into plug-and-play redistributable components with little overhead. The two most popular and recommended methods for defining tags are in-browser with the in-browser compiler, and Riot's special tag file format.

Riot Tag - *tag file format*

```
<mytag>
  <!-- HTML -->
  <h1>I am a Riot tag!</h1>
  <!-- CSS -->
```

```
  <style>
    h1 {
      color: blue;
    }
  </style>
  <!-- JavaScript -->
  <script>
    console.log('Hello world!')
  </script>
</mytag>
```

As you can see, the syntax for the tag file format is straightforward. A tag is denoted by it's given name, and its template exists anywhere within the enclosing tags that define it. Following the tag's template, styles can be defined between enclosing style tags `<style>`, and the tag's operational code can be constructed between enclosing script tags `<script>`. If this looks similar to plain old HTML, it's because it is. Riot is just plain old HTML, CSS, and JavaScript. There are no magic kicks, no directives, and no enforced semantics.

Riot Tag - *in browser definition*

```
<head>
  <title>Riot</title>
  <script src="https://rawgit.com/riot/riot/master/riot
%2Bcompiler.min.js"></script>
</head>
<body>
  <script type="riot/tag">
    <mytag>
      <!-- HTML -->
      <h1>I am a Riot tag!</h1>

      <!-- CSS -->
      <style>
        h1 {
          color: blue;
        }
      </style>

      // JavaScript Exists on Topmost Level
      console.log('Hello world!')
    </mytag>
  </script>
</body>
```

The only real difference between the tag file format and the in-browser definition, is that the tag file format encapsulates all of its operational code within enclosing `<script>` tags, whereas the in-browser definition's operational code exists mixed in with the HTML on the top-most level.

Riot's compiler also supports pre-processing for HTML, JavaScript, and CSS. This means you can use Pug, ES6, Coffeescript, TypeScript, LESS, Sass, and other popular supersets. The compiler has out-of-the box support for more mainstream renditions of JavaScript and CSS, but the compiler allows for custom parsing.

Example From Docs

```
riot.parsers.js.myJsParser = function(js, options) {
  return doYourThing(js, options)
}
```

The code sample above is run during compilation, which is covered in the following section. Note that `riot.parsers` is part of the Node specification for Riot. There are minor differences between using Riot in the browser versus the Node runtime, but the differences are primarily composed of tooling for custom configurations.

Using a tag

Before a tag can be used it must be transformed into JavaScript for Riot to consume. This can be done using the Riot compiler--the maintainers of Riot officially support loaders for popular build tools including Webpack, Browserify, and Gulp. As mentioned in the section above, Riot also supports in-browser compilation, which is not suggested for medium to large projects because of the workload that would be offloaded to the end user's browser, but it's certainly a viable solution for building simple widgets.

Once the tag has been compiled into a consumable format, the tag must be registered using the `riot.tag(...)` API method. The loaders mentioned will automatically register your tag for you. Using `riot.tag` you may also manually construct your own tags without the need of the compiler; this can be especially helpful for partial tags.

Registering Tags

```
// riot.tag(tagName, html, [css], [attrs], [constructor])
riot.tag('mytag', '<h1>I am a Riot tag!</h1>', 'h1 { color:
blue; }', '', function() {
```

```
  console.log('Hello World')
})
```

The example above is what the Riot compiler would transform our first example into. `riot.tag` in its entirety is essentially the corner stone of Riot's component architecture. After a tag has been registered, the tag can be rendered into your view like so:

```
// riot.mount(tagName)
riot.mount('mytag')
```

When using the in-browser compiler, just like the loaders, the compiler will automatically take care of registering tags.

Mounting Tags - *(Runnable) Example Using Web Compiler*

```
<head>
  <title>Riot</title>
  <script src="https://rawgit.com/riot/riot/master/riot
%2Bcompiler.min.js"></script>
</head>
<body>
  <mytag />
  <script type="riot/tag">
    <mytag>
      <h1>Hello!</h1>
    </mytag>
  </script>
  <script>
    // no need to register the tag
    riot.mount('mytag')
  </script>
</body>
```

The Riot compiler is included with the official NPM package, but the Riot team also provides a browser rendition that releases, which can be accessed via Bower or various Javascript CDNs. An important note to consider is that the compiler is one of the discrepancies mentioned in the previous section between both the browser and the Node runtime. The in-browser compiler will automatically compile and register any tags wrapped within `<script type="riot/tag">`, but it can also be leveraged to load tags from external URLs. When loading tags from external URLs, the compiler expects tags in the tag file format mentioned in **Creating a Tag**. Though the compiler is agnostic to the file type, Riot's coined file extension for tags is `.tag`.

```
// riot.compile(url, callback)
riot.compile('mysite.com/foobar.tag')
```

Using Riot in the Node runtime gives you the power to create your own build pipelines for the tags. This can be incredibly helpful for creating a lightweight, minimal footprint build process rather than relying on larger build tools such as Webpack.

```
const riot = require('riot-compiler')
const fs = require('fs')
const tagPath = './src/components/component.tag'
const tagSource = fs.readFileSync(tagPath, 'utf8')
const options = {}

const js = riot.compile(tagSource, options, tagPath)
```

You can also provide your own parsers as also mentioned in the previous section:

```
const coffee = require('coffeescript')
const riot = require('riot-compiler')
const fs = require('fs')
const tagPath = './src/components/component.tag'
const tagSource = fs.readFileSync(tagPath, 'utf8')
const options = {}

const js = riot.compile(tagSource, options, tagPath)

riot.parsers.js.coffeeParser = function(js, options) {
  return coffee.eval(js).toString()
}

const js = riot.compile(tagSource, options, tagPath)
```

Finally, the official NPM package includes the Riot CLI as a dependency, which can be used to quickly pre-compile your tags.

```
# install riot globally with npm
npm install -g riot

# compile tag
riot src/components/component.tag
```

If you've made it this far, congratulations. You now understand the anatomy of Riot's component architecture. Yes, it's that simple!

Expressions

Expressions are where all of Riot's magic happens. Underneath the hood, Riot caches tag templates and only re-renders expressions whenever an updated occurs to help increase performance. Expressions are just JavaScript, but proper use can make or break your components.

Interpolation

```
<head>
  <title>Riot</title>
  <script src="https://rawgit.com/riot/riot/master/riot
%2Bcompiler.min.js"></script>
</head>
<body>
  <script type="riot/tag">
    <date>
      <h1>
        <!-- contents of the expression in this context are
evaluated and returned -->
        Today's Date: { new Date().toLocaleString() }
      </h1>
    </date>
  </script>
</body>
```

As seen in the example above, expressions are defined by two enclosing curly brackets. Any content within the curly brackets is evaluated whenever an update is emitted by your component. Expressions have a variety of use cases, from binding properties and events, to dynamically constructing DOM attributes.

Dynamic Class Names and Attributes

```
<date>
  <!-- will be evaluated to <h1 class="danger"> -->
  <h1 class={ danger: true, info: false }>
    <!-- contents of the expression in this context are
evaluated and returned -->
    Today's Date: { new Date().toLocaleString() }
    <!-- will be evaluated to <input type-"checkbox"
checked /> -->
    <input type="checkbox" { true ? 'checked' : '' } />
  </h1>
</date>
```

Loops and Conditions

Using expressions you can dynamically hydrate your components, but Riot also provides data centric paradigms for loops and conditions. Conditions are quite straightforward:

```
<head>
  <title>Riot</title>
  <script src="https://rawgit.com/riot/riot/master/riot
%2Bcompiler.min.js"></script>
</head>
<body>
  <script type="riot/tag">
    <hidden>
     <!-- element will not be rendered on DOM -->
      <h1 if={ visible }>Hello World!</h1>

      // within the scope of your tag's operational code
      // "this" can be used to access the tag's instance
      this.visible = false
    </hidden>
  </script>
</body>
```

The `if` attribute takes a truthy value in the form of an expression, and simply toggles the given element either visible or hidden using the CSS display property. Loops are also quite simple to use:

```
<head>
  <title>Riot</title>
  <script src="https://rawgit.com/riot/riot/master/riot
%2Bcompiler.min.js"></script>
</head>
<body>
  <script type="riot/tag">
    <fruits>

      <ul>
        <!-- without an alias, the intermediate value can be
accessed by the property item -->
        <li each={ fruits }>{ item }</li>
      </ul>

      <ul>
        <!-- intermediate variables can be given an alias
for readability -->
```

```
        <li each={ vegetable in vegetables }>{ vegetable }</
li>
      </ul>

      this.fruits = ['banana', 'cheesecake', 'potato']
      this.vegetables = ['carrots', 'chipotle', 'bricks']

    </fruits>
  </script>
</body>
```

Loops are denoted by an attribute `each` that consumes an iterable object as an expression. Intermediate variables can be accessed within the context of the loop by their given alias, otherwise Riot will yield the value to a property `item`. A very important note to make is `each` will create a new context that may override your tag's instance namespace. To ensure you're fetching data from the tag instance within the context of the loop, be sure to fetch your given data from the immutable `parent` property.

```
<head>
  <title>Riot</title>
  <script src="https://rawgit.com/riot/riot/master/riot
%2Bcompiler.min.js"></script>
</head>
<body>
  <script type="riot/tag">
    <lineup>
      <ul>
        <!-- in this context, the property "hero" from the
tag instance will not be directly available -->
        <!-- we can access the tag's hero property using
"parent" -->
        <li each={ hero in heros }>{ parent.hero } vs
{ hero }</li>
      </ul>

      this.hero = 'saitama'
      this.heros = ['superman', 'goku', 'mumen rider']
    </lineup>
  </script>
</body>
```

Virtual and yield

Both looping and dealing with conditions in Riot is rudimentary, but with more advanced layouts you may find yourself in a pinch where you don't necessarily require a parent wrapper. Riot offers the `virtual` tag as a catalyst for rendering both loops and conditions, which will be removed by Riot when a component is rendered.

```
<head>
  <title>Riot</title>
  <script src="https://rawgit.com/riot/riot/master/riot
%2Bcompiler.min.js"></script>
</head>
<body>
  <script type="riot/tag">
    <doggochart>
      <!-- outermost virtual tag will not be rendered -->
      <virtual if={ true }>
        <dl>
          <!-- outermost virtual tag will not be rendered -->
          <virtual each={ doggo in doggos }>
            <dt>{ doggo }</dt>
            <dd>is still a doggo</dd>
          </virtual>
        </dl>
      </virtual>

      this.doggos = ['happy doggo', 'sad doggo', 'ninja
doggo']
    </doggochart>
  </script>
</body>
```

Additionally, Riot supports "HTML transclusion," or more simply put, nesting HTML, for more control in intricate layouts in the form of the `yield` tag. Though a commonly overlooked feature, the `yield` tag allows developers to interpolate components and content with ease. This can be helpful for constructing data driven layouts.

```
<head>
  <title>Riot</title>
  <script src="https://rawgit.com/riot/riot/master/riot
%2Bcompiler.min.js"></script>
</head>
<body>
  <!--
```

```
    the following html could be rendered as:
    <message>
      <span>
        Hello, Robin
      </span>
    </message>
  -->
  <message>Robin</message>

  <script type="riot/tag">

    <message>
      <span>
        Hello, <yield />
      </span>
    </message>

  </script>
</body>
```

You can even yield components:

```
<head>
  <title>Riot</title>
  <script src="https://rawgit.com/riot/riot/master/riot
%2Bcompiler.min.js"></script>
</head>
<body>
  <!--
    the following html could be rendered as:

    <fits>
      <span>
        If it fits, <sits><span>I sits</span></sits>
      </span>
    </fits>
  -->
  <fits>
    <sits />
  </fits>

  <script type="riot/tag">

    <fits>
      <span>
        If it fits, <yield />
      </span>
    </fits>
```

```
    <sits>
      <span>I sits</span>
    </sits>

  </script>
</body>
```

DOM events

Riot 3 ships with a powerful and incredibly flexible internal event handler that supports DOM event model level 3 events. Event callbacks can easily be bound to elements defined within tags by their event types using expressions:

```
<head>
  <title>Riot</title>
  <script src="https://rawgit.com/riot/riot/master/riot
%2Bcompiler.min.js"></script>
</head>
<body>
  <script type="riot/tag">
    <alert>
      <!-- we can bind children of the tag instance -->
      <button click={ alert }>Click Me!</button>

      alert(e) {
        // the riot compiler will automatically transform
es6 style functions into
        // children of the corresponding tag instance
        window.alert('Hey!')
      }
    </alert>
  </script>
</body>
```

Whenever the button in the sample above is clicked, Riot's event dispatcher will execute the `alert` method and provide an object with event details. The object provided contains all of the same properties of a typical DOM event object, with the addition of a property `item`, which can be used to pull any other data bound to a given element. The `item` property can be especially useful in loops.

```
<head>
  <title>Riot</title>
```

```
  <script src="https://rawgit.com/riot/riot/master/riot
%2Bcompiler.min.js"></script>
</head>
<body>
  <script type="riot/tag">
    <menu>
      <ul>
        <li each={ pizza in pizzas }>
          <!-- we can bind intermediate data to our
generated elements -->
          <button click={ parent.purchase }
pizza={ pizza }>{ pizza.name }</button>
        </li>
      </ul>

      this.pizzas = [
        {
          name: 'Cheese Pizza',
          price: 1.00
        },
        {
          name: 'Bacon Pizza',
          price: 1337.00
        }
      ]

      purchase(e) {
        // we can pull any bound expressions using the item
property from the event
        const pizza = e.item.pizza
        window.alert(`You have chosen the ${ pizza.name }`)
        window.alert(`Your total is $${ pizza.price }, now
pay up!`)
      }
    </menu>
  </script>
</body>
```

Something to be mindful of, is that by design any events dispatched to the Riot event handler will automatically trigger an update on attributed components. Riot, however, does allow you to intercept the update. This can be done by toggling the property `preventUpdate` via the provided event object mentioned above.

```
<head>
  <title>Riot</title>
  <script src="https://rawgit.com/riot/riot/master/riot
```

```
%2Bcompiler.min.js"></script>
</head>
<body>
  <script type="riot/tag">
    <time>
      <!-- expression will evaluate to initial value on
render -->
      <h1>Current Time: { time }</h1>

      <!-- clicking this button will trigger a component
update -->
      <!-- upon the update, the expression above will
evaluate the new value of the property "time" -->
      <button click={ updateTime }>Update Time</button>

      <!-- clicking this button will still update the tag
instance's property "time" -->
      <!-- riot will not re-evaluate the expression above -->
      <button click={ updateTimeNoUpdate }>Update Time (No
Update)</button>

      // defining initial value of property time
      this.time = new Date().toLocaleString()

      updateTime(e) {
        this.time = new Date().toLocaleString()
      }

      updateTimeNoUpdate(e) {
        this.time = new Date().toLocaleString()
        // by toggling preventUpdate, the component update
can be averted
        e.preventUpdate = true
      }
    </time>
  </script>
</body>
```

`preventUpdate` can be helpful in situations where you may want to update your data, but don't necessarily want Riot to re-evaluate your component. A note worth making is that when a component is prompted for an update, each of the component's child components will subsequently also be re-evaluated. In larger applications that can be very taxing in performance.

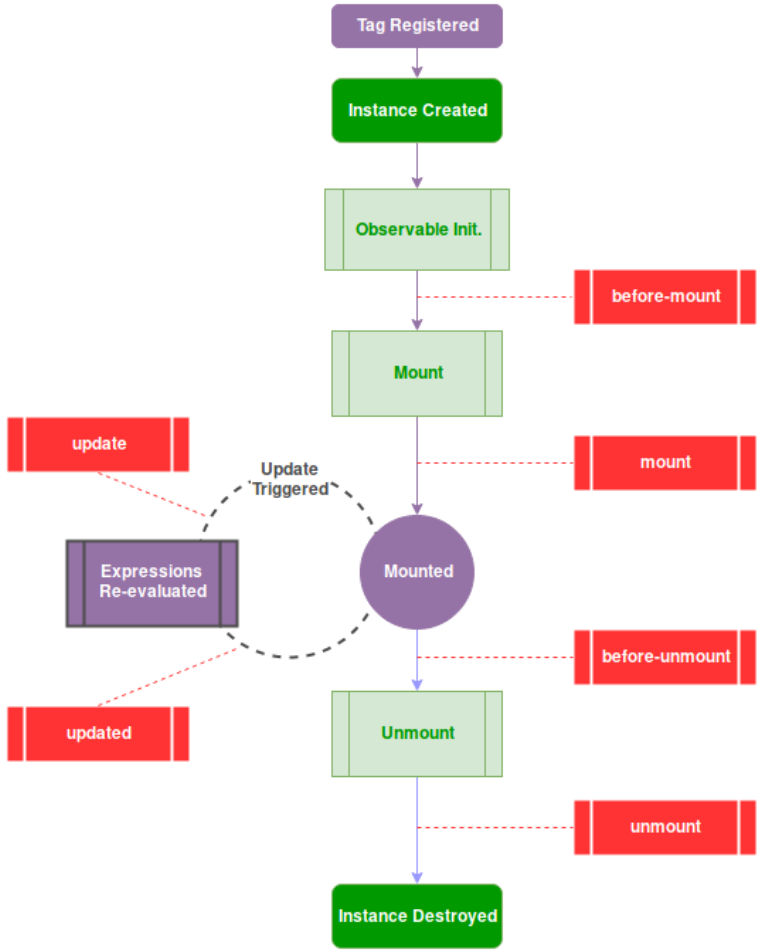
Tag lifecycle

Like many other popular JavaScript frameworks, Riot components offer a number of lifecycle events to help keep your interface reactive and easy to manage. Riot's lifecycle events are relatively straightforward, as well as incredibly easy to hook into. These lifecycle events are documented as:

```
<todo>

  <script>
    this.on('before-mount', function() {
      // before the tag is mounted
    })

    this.on('mount', function() {
      // right after the tag is mounted on the page
    })

    this.on('update', function() {
      // allows recalculation of context data before the
update
    })

    this.on('updated', function() {
      // right after the tag template is updated after an
update call
    })

    this.on('before-unmount', function() {
      // before the tag is removed
    })

    this.on('unmount', function() {
      // when the tag is removed from the page
    })

    // curious about all events ?
    this.on('*', function(eventName) {
      console.info(eventName)
    })
  </script>

</todo>
```

The `mount` and `update` events should seem familiar from the information covered earlier in this chapter. These hooks provide you with complete control over each phase of a given component's lifecycle.

Riot Lifecycle Events



The observable

As shown in the previous figure, prior to the lifecycle hooks, yet after the component has been created, the observable is instantiated. The observable is one of the more unique aspects of Riot, and most certainly

one of the most powerful. It provides a push like or subscribe and publish mechanism for seamlessly sending and receiving events. The observable is also actually a generic tool that can be used outside of the scope of a Riot component, so any object can be turned into an observable using `riot.observable`.

Node Runtime

```
const riot = require('riot')

class Programmer {
  constructor() {
    // riot.observable(target)
    riot.observable(this)
  }
  code() {
    this.trigger('coding')
  }
  sleep() {
    const self = this
    self.trigger('sleeping')
  }
}

const programmer = new Programmer()

programmer.on('coding', function() {
  // will be executed each time `programer.code` is called
})

programmer.one('sleeping', function() {
  // will be executed once, when `programmer.sleep` is called
})
```

In the example above, a non-ui, non-riot specific entity is turned into an observable. The observable API consists of just four methods: `on`, `one`, `off`, and `trigger`, but it drives the entire component lifecycle architecture. Let's look back at the documented hooks for the lifecycle events,

```
this.on('update', function() {
  // allows recalculation of context data before the update
})
```

This “hook” is just tuning into the tag instance’s observable, but there’s nothing special going on underneath the hood--it’s really that simple.

Mixins

Mixins are exactly what they sound like. They allow you to “mix in” code into your tag. What this means is that you can write a single mixin and use it across your tags.

If you need to share code across tags, mixins are the way to go. In our example app, we have a global mixin registered in `app.js` (**https://github.com/backstopmedia/riot-book-example/blob/master/src/app.js**):

Global mixin

```
import Tracker from '@/services/tracker.js'

// # install tracker service as a named global mixin
riot.mixin({ tracker: new Tracker(riot) })
```

This allows you to use the tracker service in `services/tracker.js` in all of the tags.

You can also register mixins in individual tags if you would like to share code between certain tags, but not all tags.

Individually registered mixin

```
class Mixin {
    ...
}

<riot-mixin-example>
...

<script>
    this.mixin(Mixin)
</script>
<riot-mixin-example>
```

Here is an example from our DevOps dashboard app of registering a mixin within a tag:

```
components/service-card.tag
```

```
<ServiceCard>
    ...
    <script type="es6">
        import Chart from 'chart.js'
        import time from '../mixins/time'
        const self = this

        ...

        self.mixin(time)
    </script>
</ServiceCard>
```

And the mixin itself:

```
mixins/time.js

export default {
  /**
   * Convert seconds to minutes
   * @param {Int} seconds - seconds to convert.
   */
  secondsToMinutes(seconds) {
    return Math.floor(seconds / 60)
  }
}
```

The advantage of doing this is that you can share mixins and only load the mixin where you need it.

Alternatively, you can share mixins by registering a mixin with a key and using it like this:

Registering a mixin with a key

```
mixin.js

class Mixin {
    ...
}

riot.mixin('ourMixin', Mixin)

riot-mixin-example.tag

<riot-mixin-example>
    ...

    <script>
```

```
        this.mixin('ourMixin')
    </script>
</riot-mixin-example>
```

Using a key allows you to use this mixin in any tag without having to import it in each tag where you want to use it.

Putting it all together

Let's walk through building a tag together. For simplicity, we will leave out mixins and observables for now because we only want to create one tag. Later chapters will cover how to use them in a real world setting.

Every good app needs some sort of header or navigation bar, so let's build one for our dashboard. Let's write out the HTML first so you know the general structure of the tag. Our header is a Bootstrap navbar so the `navbar-*` classes should be familiar if you've used Bootstrap (**https://getbootstrap.com**) before, and our `fa-github` icon at the end of the HTML is a GitHub icon from the Font Awesome library (**http://fontawesome.io/**).

components/header.tag

```
<Header>
  <nav class="navbar has-shadow" role="navigation" aria-
label="main navigation">
    <div class="navbar-brand">
      <a class="navbar-item" href="/">
        <img src=""
             alt="Bleeding Edge Press: Publishing at the
speed of technology"
             width="112"
             height="28">
      </a>
    </div>
    <div class="navbar-menu">
      <div class="navbar-start">
          <a class="navbar-item is-tab is-active" href='#'>
            A Link
          </a>
      </div>
      <div class="navbar-end">
        <a class="navbar-item" href="https://github.com/
backstopmedia/riot-book-example" target="_blank">
          Fork on Github
          <span class="icon is-large">
```

```
            <i class="fa fa-2x fa-github"></i>
          </span>
        </a>
      </div>
    </div>
  </nav>
</Header>
```

Let's start getting some data into our tag, so we can work on the JavaScript part now.

components/header.tag

```
<Header>
    <!-- Our HTML -->
    ...

    <script type="es6">
        const self = this

        self.routes = [
          {
            name: '',
            label: 'Home',
            active: false
          },
          {
            name: 'about',
            label: 'About',
            active: false
          },
          {
            name: 'help',
            label: 'Help',
            active: false
          }
        ]

        self.brandImg = require('../assets/images/bep.png')
  </script>
</Header>
```

Our site logo should now appear as `{brandImg}` and has now been set as `require('../assets/image/bep.png')`, which will pull in `bep.png`.

Our app has more than one page, so we need that to reflect in our navbar. We can do this using an `each` loop inside of a `virtual` tag.

components/header.tag

```
<Header>
    <nav class="navbar has-shadow" role="navigation" aria-
label="main navigation">
        <div class="navbar-brand">
            <a class="navbar-item" href="/">
                <img src=""
                     alt="Bleeding Edge Press: Publishing at
the speed of technology"
                     width="112"
                     height="28">
            </a>
        </div>
        <div class="navbar-menu">
            <div class="navbar-start">
                <!-- New part -->
                <virtual each={route in routes}>
                    <a class="navbar-item is-tab {is-active:
route.active}" href='#{ route.name }'>
                        { route.label }
                    </a>
                </virtual>
                <!-- End of new part -->
            </div>
            <div class="navbar-end">
                <a class="navbar-item" href="https://
github.com/backstopmedia/riot-book-example" target="_blank">
                    Fork on Github
                    <span class="icon is-large">
                        <i class="fa fa-2x fa-github"></i>
                    </span>
                </a>
            </div>
        </div>
    </nav>

    <script type="es6">
        const self = this

        self.routes = [
          {
            name: '',
            label: 'Home',
            active: false
          },
          {
            name: 'about',
            label: 'About',
```

```
            active: false
          },
          {
            name: 'help',
            label: 'Help',
            active: false
          }
        ]

        self.brandImg = require('../assets/images/bep.png')
    </script>
</Header>
```

Remember, `<virtual>` is used for loops in Riot because it does not show up in the DOM, so your loop generated elements will not be wrapped in a container tag.

`each={route in routes}` loops through the `self.routes` array. From there, you simply access object properties as usual and assign the `href` and link names.

There is also usage of a conditional on the `<a>` tag to conditionally assign the `is-active` class so only the currently active link has a different text color and underline. If `route.active` equals true, then the `is-active` class will be added to the element.

The example app uses the `riot-route` package, which is the default Riot router. You can incorporate this to be able to change the value of `route.active` and set an active route when clicking on one of the navbar links. We will not cover how the router works in the chapter, but it will be explained in later chapters as we walk you through how to put all of these techniques together to build a full Riot app.

```
header.tag
<Header>
    <!-- Our HTML -->
    ...

    <script type="es6">
        const self = this

        self.routes = [
          {
            name: '',
            label: 'Home',
            active: false
          },
          {
```

```
            name: 'about',
            label: 'About',
            active: false
          },
          {
            name: 'help',
            label: 'Help',
            active: false
          }
        ]

        // # router middleware to track active route
        route(function(target, action, params) {
            if (self.routes) {
                let previous = self.routes.find(r =>
r.active)
                if (previous)
                    previous.active = false
                    self.routes.find(r => r.name ==
target).active = true
                    self.update()
            }
        })

        self.brandImg = require('../assets/images/bep.png')
  </script>
```

Summary

Congratulations! As you can see, Riot is a very simple library and has a low learning curve. You are now equipped with the knowledge to get started building a full Riot application.