Chapter 5. Routing and Pathing

5.1. Definition

Larger applications in Backbone are going to have routes. You can get a ways with just putting strings everywhere, but after a while this becomes painful when a route changes, or when you have particularly complex routes.

First off, let's get our definitions out of the way. A route is what a router uses to determine what to do given the value of the browser window's location. A Backbone route is what you put in a router's definition, and looks like `/articles/:id/edit`.

Next, a path is the url for a given resource, and it should match a route in our application. For example, a path to the above route for an article with an id of 4 would be `/articles/4/edit`. I'm also going to use the word path to describe a function that returns a url. For example `editArticlePath(4) => /articles/4/edit`.

Routes and paths work together, in that a path is handled by a route. Let's look at some specs for paths and routes:

```
describe "Routes", ->
  describe "articles", ->
    it "should have an index route", ->
      expect(mb.a.Routes.articles).toEqual "articles"
    it "should have a resource route", ->
      expect(mb.a.Routes.article).toEqual "articles/:id"
    it "should have an edit route", ->
      expect(mb.a.Routes.editArticle).toEqual "articles/:id/edit"
    it "should have a new route", ->
      expect(mb.a.Routes.newArticle).toEqual "articles/new"

describe "Paths", ->
  describe "articles", ->
    it "should have an index path", ->
      expect(mb.a.Paths.articles()).toEqual "/articles"
    it "should have a resource path", ->
      expect(mb.a.Paths.article(4)).toEqual "/articles/4"
    it "should have an edit path", ->
      expect(mb.a.Paths.editArticle(4)).toEqual "/articles/4/edit"
    it "should have a new path", ->
      expect(mb.a.Paths.newArticle()).toEqual "/articles/new"
```

We've got our basic set of RESTful routes (plus new and edit) that are suitable for use in our router. We've also got pathing functions that could be used in views or in event bindings to create links or navigate based on actions.

Notice that even the paths that do not take parameters are functions. This is so that the `Paths` object always presents a consistent interface. Otherwise you'd have to switch between attributes and functions based on the path, which is confusing.

Here's the implementation:

```
mb.a.Routes = Routes =
  articles:    "articles"
  newArticle:  "articles/new"
  article:     "articles/:id"
  editArticle: "articles/:id/edit"

mb.a.root = root = "/"

mb.a.Paths =
  articles:         -> root + Routes.articles
  newArticle:       -> root + Routes.newArticle
  article:     (id) -> root + Routes.article.replace(":id", id)
  editArticle: (id) -> root + Routes.editArticle.replace(":id", id)
```

Note

I probably wouldn't unit test my routes, they are very simple. I probably wouldn't test paths either, except maybe some very complicated ones, for example using a wildcard path.

5.2. Routes and the Router

The only place to use routes in the application is the router. If that's the only place, you may wonder why we don't just code the routes into the router directly. I like to keep information in an application as close as possible to other related information. My perspective is that routes and paths are more closely related than the routes and the router. It would also mean that I'd have to pull routes out of the router in order to make my paths. routes and paths are also at a higher level in the application than the router. routes and paths are part of the core of your application, but the router is only there to turn routes into actions. Therefore it's helpful to have the router access the routes in order to figure out what it requires to route.

So, we'd like our router to use our routes in order to navigate, like this:

```
describe "Router", ->
  describe "articles", ->
    router = null
    spy    = null

    beforeEach ->
      router = new mb.a.Router()
      spy = jasmine.createSpy("RouteEvent")
      Backbone.history.start root: mb.a.root, pushState: true

    afterEach -> expect(spy).toHaveBeenCalled()

    it "should route to the index", ->
      router.on('route:articles', spy)
      mb.a.navigate(mb.a.Paths.articles())

    it "should route to a resource", ->
      router.on('route:article', spy)
      mb.a.navigate(mb.a.Paths.article(1))

    it "should route to edit", ->
      router.on('route:editArticle', spy)
      mb.a.navigate(mb.a.Paths.editArticle(1))

    it "should route to new", ->
      router.on('route:newArticle', spy)
      mb.a.navigate(mb.a.Paths.newArticle())
```

We're using Jasmine's spies to spy on the events the router fires. This is a pretty clean way to ensure the appropriate behavior, as we don't have to inspect any of the router's internal actions (what it does when it routes). We can simply rely on the built-in events to ensure it routes to the action we sent it to.

Then, for each of our paths for our article resource, we're going to bind to the routing event and then navigate. Here we've written our own navigation helper. I like to default to `trigger:true` on routing, and it's cleaner to wrap that in a helper than to send it as an option every time.

Now, let's take a look at the implementation:

```
mb.a.navigate = (fragment, options) ->
  options = _.extend {trigger: true}, options
  Backbone.history.navigate(fragment, options)

class mb.a.Router extends Backbone.Router
  initialize: ->
    for [action, route] in _(mb.a.Routes).pairs().reverse()
      @route(route, action)
```

I included the `navigate` helper here, even though that would normally be in the master application file. It simply defaults to `trigger:true` unless overridden (although that's not tested here).

The router iterates the `action` and `route` pairs from our routes definition, calling the Backbone router's `route` method to bind them together. That way the action on the router will be the same as the key in the routing table which will be the same as the method on the `Paths` object.

One thing you may notice here that's a little odd is the `reverse` on the routes. When you define a router's routes via the `routes` key in the class definition, Backbone does a reverse on them as well. This is to get the priorities correct. The **last** route specified takes priority, which is counter-intuitive to the way we would define routes, as we'd expect the highest route to take priority. So, since `articles/new` matches both the `articles/new` route and the `articles/:id` route, we want to specify `articles/new` first, which has to actually be called last, hence the reverse. A bit annoying, but that's why we have tests to catch these issues.

Note

I didn't actually define the actions here. In a real application, our next step would be to define the routing actions, but we didn't need to in order to ensure our routing was working properly, as events are still fired.

5.3. Paths and the View

OK, our router is setup with our routes, now it's time to use our paths to drive our application. There are two common places to use paths: links and event bindings. Both of these are in Views (or via the View's template).

Let's look at how we'd have an `ArticleView` with an `edit` link that links to that article's edit url and a delete button that deletes the article and redirects to the articles path:

```
describe "ArticleView", ->
  model = null
  view  = null

  beforeEach ->
    model = new Backbone.Model { id: 4 }
    view = new mb.a.ArticleView model: model
    view.render()

  it "has a link to its edit view", ->
    link = view.$el.find('a').attr 'href'
    expect(link).toEqual mb.a.Paths.editArticle(model.id)

  it "redirects to the index page after being destroyed", ->
    spyOn model, 'destroy'
    spyOn mb.a,  'navigate'
    view.$el.find('button.delete').click()
    expect(model.destroy).toHaveBeenCalled()
    expect(mb.a.navigate).toHaveBeenCalledWith(mb.a.Paths.articles())
```

We setup a fake model, give it to the view and render it, then we check the link the view creates to make sure it paths correctly. This is a relatively invasive test, and it depends on the view to create a link, but that is what we are trying to ensure at this stage.

Note

`a` tags are HTML tags that lead to urls, and that should be the case in Backbone apps as well! It may be tempting to use

a button and a click binding instead of an `a` tag because it's easier up front, but this can be pretty annoying, especially for screen readers or people crawling your site. Later on we'll show how to capture link navigations and turn them into Backbone navigations.

For the delete step, we have a button that we listen to for the delete action. Delete isn't a url or a destination, it's an action we need to perform, that's why we'll use events. We'll expect the model's destroy method to be called, and we'll also expect a navigation to our articles path. Here's another great reason to wrap Backbone's navigate: we're mocking against our helper instead of Backbone, and you should only mock objects you own!

Here's the implementation for the view:

```
class mb.a.ArticleView extends Backbone.View
  events:
    "click button.delete": "delete"

  template: Mustache.compile """
    <a href="{{ editLink }}">Edit</a>
    <button class="delete">Delete</button>
  """

  render: ->
    @$el.html(@template(this))
    this

  editLink: =>
    mb.a.Paths.editArticle(@model.id)

  delete: ->
    @model.destroy()
    mb.a.navigate mb.a.Paths.articles()
```

In this view we're using Mustache and a CoffeeScript heredoc (multiline escaped string). I really like Mustache, especially for the use case shown here. Mustache is an intelligent template library, and it will notice that `editLink` is a function and automatically call it. I prefer to pass the view itself as the context to the template (and not the model), and then only

access methods on the view. We'll see this technique more in Model and View Accessors. In this situation, I've got a view helper to build the edit link, and I can call it easily from the template.

Other than that, this should look like a standard Backbone view. The best part: there are no raw urls anywhere! It's all wrapped up in helpers. Nice and clean and tested.