Filled with real-world JavaScript scaling scenarios and code-first examples, JavaScript at Scale is your guide to building applications that last and will show you how to deal with scalability from a number of perspectives: addressability, testability, and component composition. By understanding the fundamentals of scaling issues, you'll be able to use that knowledge to tackle even the most difficult situations.

The book begins by defining "scale" from a JavaScript point of view, and further dives into the influencers of scale, as well as scalable component composition and communication. Large-scale architectures need the ability to scale down, and recover from failing components, as well as scale up and manage new features or a large user base. From here, you can build applications that scale using any set of JavaScript tools.

Who this book is written for
Have you ever come up against an application that felt like it was built on sand? Maybe you've been tasked with creating an application that needs to last longer than a year before a complete re-write? If so, JavaScript at Scale is your missing documentation for maintaining scalable architectures. Most concepts presented are adaptations of components found in frameworks such as Backbone, Angular, or Ember.

What you will learn from this book
- Identify and evaluate the external scaling influencers of your application
- Build out component composition in large-scale JavaScript applications
- Discover the techniques for inter-component communication that scale
- Customize and configure components to meet scaling demands
- Ensure a highly-performant user experience, despite the size and complexity of your application
- Create a portable and testable application that's not constrained to one environment
- Make architectural trade-offs by removing or refactoring components
- Design components that fail in a scalable way
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 5 'Addressability and Navigation'
- A synopsis of the book’s content
- More information on JavaScript at Scale
Adam Boduch has been involved with large-scale JavaScript development for nearly 10 years. Before moving to the frontend, he worked on several large-scale cloud computing products using Python and Linux. No stranger to complexity, Adam has practical experience with real-world software systems and the scaling challenges they pose. He is the author of several JavaScript books, including Lo-Dash Essentials, and is passionate about innovative user experiences and high performance.

Adam lives in Toronto and is a senior software engineer at Virtustream.
Preface

Some applications just get it right. These are the exceptions rather than the rule. Lots of JavaScript applications get one or two things right, and other things very wrong. The things we get wrong are a side effect of the scaling influencers that we never considered. This is a book about scaling our frontend architectures to meet the quality requirements asked of us. Scaling JavaScript applications is an interesting and fun problem. There’re so many moving parts—the users, the developers, the deployment environments, the browser environments, and the task of bringing all of these factors together to form a meaningful user experience. What are we scaling, and why? The aim of this book is to help us answer these questions.

What this book covers

Chapter 1, Scale from a JavaScript Perspective, introduces the idea of scalable JavaScript applications and what makes them different from other applications that scale.

Chapter 2, Influencers of Scale, helps us understand that the need to scale helps us design better architectures.

Chapter 3, Component Composition, explains how the patterns that form the core of our architecture serve as blueprints for assembling components.

Chapter 4, Component Communication and Responsibilities, explains how components that communicate with one another are a scaling constraint. It tells us how features are the result of component communication patterns.

Chapter 5, Addressability and Navigation, elaborates on large-scale web applications with URLs that point to resources, and how designs that scale can handle a growing number of URLs.
Chapter 6, User Preferences and Defaults, tells us why users need control over certain aspects of our software. And it also explains that scalable application components are configurable.

Chapter 7, Load Time and Responsiveness, explains how more moving parts means performance degradation across the application. This includes making trade-offs that keep our UI responsive, while adding new features.

Chapter 8, Portability and Testing, covers writing JavaScript code that's not tightly coupled with a single environment. This includes creating portable mock data and portable tests.

Chapter 9, Scaling Down, explains how removing unused or buggy components from applications is essential, if we want to scale up in other areas.

Chapter 10, Coping with Failure, explains that large-scale JavaScript architectures can't fall over as a result of a bug in one component. This includes how designing with failure in mind is the key to achieving scale in a broad number of scenarios.
Addressability and Navigation

Applications that live on the web rely on addressable resources. The URI is an essential internet technology. It eliminates a whole class of complexity, because we can encode bits of information about resources into URI strings. That's the policy part. The mechanism part is up to the browser, or our JavaScript code—looking up the requested resource and displaying it.

In the past, processing URIs took place in the backend. The browser's responsibility, when the user passed it a URI, was to send this request to the backend and display the response. With large-scale JavaScript applications, this responsibility has shifted mostly to the frontend. We have the tools to implement sophisticated routing in the browser, and with that, there's less reliance on backend-technologies.

The benefits of frontend routing do come at a cost, however, once our software packs on features. This chapter takes a deep look into the routing scenarios that we're likely to encounter as our application architecture grows and matures. Most low-level implementation specifics of router components from frameworks, aren't important. We're more concerned with how well our router components adapt to scaling influencers.

Approaches to routing

There are two approaches to routing in JavaScript. The first is using hash-based URIs. These are the URIs that begin with the # character and this is the more popular approach. The other less popular approach is to use the history API of the browser to generate more traditional URIs the web population is used to. This technique is more involved, and has only recently gained enough browser support to make it viable.
Hash URIs

The hash portion of the URI was originally intended to point to a specific location in the document. So the browser would look at everything to the left of the # character, and send this information to the backend, asking for some page content. Only when the page arrived and was rendered did the right side of the # character become relevant. This is when the browser used the hash portion of the URI to find the locally relevant spot within the page.

Today, the hash portion of the URI is used differently. It's still used to avoid passing irrelevant data to the backend when the URI changes. The main difference is that today we're dealing with applications and features instead of web sites and static content. Since most of the application is already loaded into the browser when the address changes, it doesn't make sense to send unnecessary requests to backend. We only want the data that we need for the new URI, and that's usually accomplished with an API request in the background.

When we talk about using the hash approach to URIs in JavaScript applications and changing the URI, it's usually only the hash portion that changes. This means that the relevant browser events will fire, notifying our code that the URI changed. But it won't automatically issue a request to the backend for new page content, and this is key. We can actually get a lot of performance and efficiency gains out of frontend routing like this, and that's one of the reasons we use this approach.

Not only does it work well, but it's easy to implement. There's not a lot of moving parts in implementing a hash change event listener that executes logic to fetch the relevant data, and then updates the page with the relevant content. Further, the browser history changes are automatically handled for us.

Traditional URIs

For some users and developers, the hash approach just feels like a hack. Not to mention the SEO challenges presented in a public internet setting. They prefer the look and feel of the more traditional slash-separated resource name format. That's generally possible to achieve now in all modern browsers, thanks to enhancements to the history API. Essentially, the routing mechanism can listen for states being pushed onto the history stack, and when that happens, it prevents the request from being sent to the backend, and instead processes it locally.
There's obviously more code required for this approach to work, and more edge cases to think about. For example, the backend needs to support all the URIs that the frontend router does, because the user can feed any valid URI into the application. One technique to deal with this is a rewrite rule on the server that redirects 404 errors back to the application index page, where our real route processing lives.

That said, the router components found in most JavaScript application frameworks abstract the differences in approach and provide a means to seamlessly go in one direction or another. Does it matter which one is used, either for enhanced functionality or improved scalability? Not really. But in terms of scalability, it's important to acknowledge that there are in fact two main approaches, and that we don't want to commit ourselves entirely to one over the other.

How routers work

Now it's time for us to dig a little deeper into routers. We want to know the responsibilities of a router, and what it's lifecycle looks like when the URI changes. Essentially, this amounts to the router taking the new URI and figuring out if it's something the router is interested in. If it is, then it triggers the appropriate route events with the parsed URI data as arguments.

Understanding the role of routers at a low-level is important for scaling our application because the more URIs we have, and the more components we have responding to these route events, the more potential for scaling issues. When we know what's happening with the router lifecycle, we can make the appropriate scaling trade-offs in response to scaling influencers.

Router responsibilities

A simplistic view of a router is just a map—there's routes, string or regular expression pattern definitions, which map to callback functions. What's important is that this process is fast, predictable, and stable. This is challenging to get right, especially as the number of URIs in our application grow. Here's a rough approximation of what any router component needs to handle:

- Storing a mapping of route patterns to their corresponding event names
- Listening to URI change events—hash change or pop state
- Performing the route pattern lookup, comparing the new URI to each mapped pattern
• When a match is found, to parse the new URI according to the pattern
• Triggering the mapped route event, passing any parsed data

The route lookup process involves a linear search through the route map to find a match. This can mean significant performance degradation when there's lots of routes defined. When the route mapping is an array of objects, it can also lead to inconsistent router performance. For example, if a route is at the end of the array, it means it's checked last and performs slowly. If it's at the beginning of the array, it performs better.

To avoid performance degradation in frequently accessed URIs, we could extend the router so that it sorts the route map array by a priority property. Another approach would involve using a trie structure, to avoid linear lookups. Of course, only consider optimizations like these if there are so many routes that the router performance is measurably poor.

The router has a lot to do when the URI changes, which is why it's important to understand the lifecycle of a given route, from the time the URI changes in the address bar, to the completion of all it's event handler functions. From a performance perspective, lots of routes can negatively impact our application. From a composition perspective, it's challenging to keep track of what components create and react to which routes. This is a little easier to handle when we know what the lifecycle of any given route looks like.

**Router events**

Once the router has found a match for the changed URI, and once it has parsed the URI according to its matching pattern, its final job is to trigger the route event. The event that's triggered is supplied as part of the mapping. The URI may encode variables, and these get parsed and passed to each router event handler as data.

Route events provide an abstraction layer, which means that components that aren't routers can trigger route events.
Most frameworks ship with router components that can directly call a function in response to a route change, instead of triggering a route event. This is actually easier, and is a more direct approach that makes sense with smaller applications. The indirection we get by triggering events from the router through the event triggering mechanism is that our components are loosely coupled to the router.

This is beneficial because different components that have no knowledge of one another can listen to the same route event. As we scale, the same routes that have been in place for a while will need to take on new responsibilities, and it's easier to add new handlers than it is to keep building upon the same function code. There's also the abstraction benefit—the components that listen to route events don't care that the event is actually triggered by a router instance. This is useful when we need a component to trigger router-like behavior, without actually having to depend on the router.

**URI parts and patterns**

With large scale JavaScript applications, a lot of thought goes into the router component. We also need to put a lot of thought into the URIs themselves. What are they composed of? Are they consistent throughout the application? What makes a bad URI? Veering in the wrong direction on any of these considerations makes scaling the addressability of our application difficult.

**Encoding information**

The point of a URI is that a client can just pass it to our application, and it contains enough information that something useful can be done with it. The simplest URI just points to a resource type, or a static location within an app—/users or /home are respective examples of these types of URIs. Using this information, our router can trigger a route event, and a callback function is triggered. These callbacks wouldn't even require any arguments—they just know what to do because there's no variability.

On the other hand, router callback functions may need a bit of context. This is when encoding information within a URI becomes important. The most common use for this is when the client asks for a specific instance of some resource, using a unique identifier. For example, users/31729. Here, the router will need to find a pattern that matches this string, and the pattern will also specify how to extract the 31729 variable. This is then passed to the callback function, which now has enough context information to perform it's task.
Addressability and Navigation

URLs can grow large and complex if we try to encode lots of information in them. An example of this would be encoding query parameters for a page that displays a grid of resources. Trying to specify all the possibilities in the route pattern is difficult and error-prone. There are bound to be changes, and unanticipated edge-cases concerning the combinations used with the variables. Some will likely be optional.

When a given URI has this much potential for complexity, it's best to keep the encoding options out of the URI pattern that's passed to the router. Instead, have the callback function look at the URI and perform further parsing to figure out the context. That keeps the route specifications neat and tidy, and the odd complex handler isolated from everything else.

For common queries, we may want to provide a simple URI for our users, especially if it's presented as a link. For example, recent posts would link to /posts/recent. The handler for this URI has a few things that it needs to figure out that would otherwise need to be encoded in the URI—such as ordering and the number of resources to fetch. Sometimes these things don't need to be included in the URI, and decisions like these can benefit both the user experience and the scalability of our code.

**Designing URIs**

Resource names are a good inspiration for the URIs we create. If the URI links to a page that displays events, it should probably start with events. Sometimes, however, the resources exposed by the backend have anything but intuitive names. Or, as an organization or an industry, we like to abbreviate certain terms. These should be avoided as well, except in the case where the context of the application provides meaning.

The inverse is true as well—adding too much meaning in the URI can actually cause confusion, if it's too verbose. This can be too verbose from the individual word point of view, or from the number of URI components point of view. To help convey structure and make it easier for human eyes to parse, URIs are usually broken down into parts. For example, the type of thing, followed by the identifier of the thing. It's not really helpful to the user to encode categorical or other tangential information in the URI—it can certainly be displayed in the UI though.

Where we can, we should be consistent. If we're limiting the number of characters for a resource name, they should all follow the same limit. If we're using slashes to separate URI parts, it should be done the same everywhere. The whole idea behind this is that it scales nicely for our users when there are a lot of URIs, as they can eventually guess what a URI for something is, without having to click on a link.
While being consistent, we sometimes want certain types of URIs to stand out. For example, when we visit a page that puts a resource in a different state, or requires input from the user, we should prefix the action with a different symbol. Let's say we're editing a task—the URI might be /tasks/131:edit. We're being consistent everywhere in our application, separating our URI components with slashes. So we could have done something like /tasks/131/edit. However, this makes it seem as though it's a different resource when really, it's the same resource as tasks/131. Only now, the UI controls are in a different state.

Following is an example that shows some regular expressions used to test routes:

```javascript
// Wildcards are used to match against parameters in URIs...
console.log('second', (/^user\/(.*)/i).exec('user/123'));
// [ 'user/123', '123' ]

// Matches against the same URI, only more restrictively...
console.log('third', (/^user\/(\d+)/i).exec('user/123'));
// [ 'user/123', '123' ]

// Doesn't match, looking for characters and we got numbers...
console.log('fourth', (/^user\/([a-z]+)/i).test('user/123'));
// false

// Matches, we got a range of characters...
console.log('fifth', (/^user\/([a-z]+)/i).exec('user/abc'));
// [ 'user/abc', 'abc' ]
```

### Mapping resources to URIs

It's time to look at URIs in action. The most common form we'll find URIs in, are as links inside our application. At least, that's the idea; to have an application that's well connected. While the router understands what to do with URIs, we are yet to look at all the places where these links need to be generated and inserted into the DOM.

There are two approaches to generate links. The first is a somewhat manual process that requires the help of template engines and utility functions. The second takes a more automated approach in an attempt to scale the manageability of many URIs.
Addressability and Navigation

Building URIs manually
If a component renders content in the DOM, it potentially builds URI strings and adds them to link elements. This is easy enough to do when there's only a handful of pages and URIs. The scaling issue here is that the page count and URI count found in JavaScript applications are complimentary—lots of URIs means lots of pages and vice-versa.

We can use the router pattern mapping configuration, the structure that specifies what URIs look like and what happens when they're activated, as a reference when implementing our views. With the help of a template engine, which most frameworks use in one form or another, we can use the template features to dynamically render links as required. Or, lacking template sophistication, we'll need a standalone utility that can generate these URI strings for us.

This gets to be challenging when there are a lot of URIs to link, and a lot of templates. We have at least some help from the template syntax, which makes building these links a little less painful. But it's still time consuming and error-prone. Additionally, we'll start to see duplicative template content, thanks to the static nature of how we build links in the templates. We need to hard-code, at the very least, the type of resource we're linking to.

Automating resource URIs
The vast majority of the resources we link to are actual resources from the API, and are represented by a model or collection in our code. That being the case, it would be nice if instead of leveraging template tools to build URIs for these resources, we could use the same function on every model or collection to build the URI. That way, any duplication in our templates associated with building URIs goes away because we only care about the abstract uri() function.

This approach, while simplifying the templates, introduces a challenge with synchronizing the model with the router. For example, the URI string that's generated by the model needs to match the pattern that the router is expecting to see. So either, the implementer needs to be disciplined enough to keep the URI generation of the model in sync with the router, or the model somehow needs to base how it generates the URI string on the pattern.
If the router uses some kind of simplified regular expression syntax for building URI patterns, it's possible to keep the `uri()` function implemented by the model automatically synced by the route definition. The challenge there is that the model needs to know about the router—which can present a dependency scaling issue—we sometimes want models and not necessarily the router. What if our model stored the URI pattern that gets registered with the router? Then it could use this pattern to generate URI strings, and it's still only ever changed in one place. Another component would then register the pattern with the router, so there's no tight coupling with the model.

Following is an example that shows how the URI strings can be encapsulated in models, away from other components:

```javascript
// router.js
import events from 'events.js';

// The router is also an event broker...
export default class Router {

  constructor() {
    this.routes = [];
  }

  add(pattern, name) {
    this.routes.push({
      pattern: new RegExp('^' + pattern.replace(/:\w+/g, '.+'),
                        '(.*)'),
      name: name
    });
  }

  start() {
    var onHashChange = () => {
      for (let route of this.routes) {
        let result = route.pattern.exec(
                               location.hash.substr(1));
        if (result) {
          events.trigger('route:' + route.name, {
            values: result.splice(1)
          });
        }
      }
    }
```

[109]
Addressability and Navigation

```javascript
function isMatch(text, id) {
  if (id) {
    return text.toLowerCase().includes(id.toLowerCase());
  }
  return false;
}

function onHashChange() {
  const hash = window.location.hash;
  const { id } = hash && isMatch(hash, id) ? decodeURI(hash.slice(1)) : {};

  if (id) {
    break;
  }
}

window.addEventListener('hashchange', onHashChange);

// model.js
export default class Model {
  constructor(pattern, id) {
    this.pattern = pattern;
    this.id = id;
  }

  // Generates the URI string for this model. The pattern is passed in as a constructor argument. This means that code that needs to generate URI strings, like DOM manipulation code, can just ask the model for the URI.
  get uri() {
    return '#' + this.pattern.replace(/:\w+/, this.id);
  }
}

// user.js
import Model from 'model.js';

export default class User extends Model {
  // The URI pattern for instances of this model is encapsulated in this static method.
  static pattern() {
    return 'user/:id';
  }

  constructor(id) {
    super(User.pattern(), id);
  }

  get id() {
    return this.match && isMatch(this.match, id) ? this.match.id : undefined;
  }

  set id(id) {
    const newMatch = isMatch(id, id) ? { id } : {};
    if (newMatch.id !== this.id) {
      this.match = newMatch;
    }
  }
}

// Test
const user = new User('123');
console.log(user.id);  // undefined
user.id = '456';
console.log(user.id);  // 456
```
// group.js
import Model from 'model.js';

export default class Group extends Model {

    // The "pattern()" method is static because
    // all instances of "Group" models will use the
    // same route pattern.
    static pattern() {
        return 'group/:id';
    }

    constructor(id) {
        super(Group.pattern(), id);
    }

}

// main.js
import Router from 'router.js';
import events from 'events.js';
import User from 'user.js';
import Group from 'group.js';

var router = new Router()

    // Add routes using the "pattern()" static method. There's
    // no need to hard-code any routes here.
    router.add(User.pattern(), 'user');
    router.add(Group.pattern(), 'group');

    // Setup functions that respond to routes...
    events.listen('route:user', (data) => {
        console.log(`User ${data.values[0]} activated`);
    });

    events.listen('route:group', (data) => {
        console.log(`Group ${data.values[0]} activated`);
    });
Addressability and Navigation

```javascript
});

// Construct new models, and user their "uri" property
// in the DOM. Again, nothing related to routing patterns
// need to be hard-coded here.
var user = new User(1);
document.querySelector('.user').href = user.uri;

var group = new Group(1);
document.querySelector('.group').href = group.uri;

router.start();
```

**Triggering routes**

The most common route trigger is in the form of a user clicking a link within our application. As discussed in the preceding section, we need to equip our link generating mechanism to handle many pages, and many URIs. Another dimension of this scaling influencer is the actual triggering actions themselves. For instance, with smaller applications, there are obviously fewer links. So this also translates to fewer click events from the user—more navigation choices means higher event triggering frequency.

It's also important to consider the lesser known navigation actors. These include redirecting the user in response to some backend task completing, or just a straight-up work-around, to get from point A to point B.

**User actions**

When the user clicks a link in our application, the browser picks this up and changes the URI. This includes the entry point into our application—maybe from another web site or from a bookmark. This is what makes links and URIs so flexible, they can come from anywhere and point to anything. It makes sense to utilize links where we can because it means that our application is well connected, and processing a URI change is something our router excels at and can handle with ease.

But there're other ways to trigger URI changes and the subsequent router workflow. For example, let's say we're on a create event form. We submit the form, and the response comes back successful—do we want to leave the user at the create event page? Or do we want to take them to the page that shows the list of events, so they can see the event they just added? In the latter case, manually changing the URI makes sense and is very easy to implement.
Redirecting users

Redirecting users to a new route as the result of a successful API response is a good example of manually triggering the router. There are several other scenarios where we would want to redirect the user from where they currently are to a new page that coincides with the activity they’re performing, or to make sure they’re simply observing the correct information.

Not all heavy processing need happen in the backend—we could be faced with a local JavaScript component that runs a process, and upon completion, we want to take the user to another page within our app.

The key idea here is that the effect is more important than the cause—we don't care so much about what causes the URI change. What really matters is the ability to use the router in unforeseen ways. As our application scales, we'll be faced with scenarios where the way out is usually by a quick and simple router hack. Having total control over the navigation of our application gives us much more control over the way our application scales.

Router configuration

The mapping of our routes to their events is often larger than the router implementation itself. That's because as our application grows and acquires more route patterns, the list of possibilities gets bigger. A lot of the time, this is an unavoidable consequence of an application that's meeting its scaling demands. The trick is to not let a large number of route declarations collapse under their own weight, and this can happen in a number of ways.

There's more than one approach to configuring the routes that a given router instance responds to. Depending on the framework we're using, the router component may have more flexibility in how they're configured than others. Generally speaking, there's the static route approach, or the event registration approach. We'll also want to consider the router's ability to disable a given route at any given time.
**Static route declarations**

Simple applications usually configure their routers with a static declaration. This usually means a mapping of route patterns to callback functions, all at router creation time. What's nice about this approach is the relative locality of all the route patterns. At a glance, we can see what's happening with our route configuration, and we don't have to go hunting for specific route. However, this doesn't work is when there are lots of routes because we have to search for them. Also, there's no separation of concerns, and this doesn't play well with developers trying to do their thing independently of each other.

**Registration events**

When there are a lot of routes to define, the focus should be on encapsulated routes— which components need these routes, and how do they tell the router about them? Well, most routers will allow us to simply call a method that lets us add a new route configuration. Then we just need to include the router and add the routes from the component.

This is definitely a step in the right direction; it allows us to keep the route declarations in the components that need them, rather than kludging together an entire applications' worth of route configurations into a single object. However, we can take this scalability a step further.

Rather than having our components directly depend on a router instance, why not trigger an add route event? This will get picked up by any router that's listening for the event. Perhaps our application is using multiple router instances, each of which have their own specializations— logging, say —and they can all listen for added routes based on specific criteria. The point is, our components shouldn't have to care about the router instance, only that something is going to trigger route events when a given pattern matches against a URI change.

![Diagram]

How to keep components isolated from routers by using events
Deactivating routes

After we've configured a given route, do we assume that it'll always be a viable route throughout the duration of the session? Or, should the router have some means to deactivate a given route? It depends on how we look at specific cases from a responsibility perspective.

For example, if something has happened, and some route should no longer be accessible—trying it just results in a user-friendly error—the route handler function can check whether the route is accessible or not. However, this adds complexity to the callback functions themselves, and this complexity will be sprinkled throughout the application in callbacks, rather than being self-contained in one place.

An alternative approach would be to have some sanity-checking component that deactivates routes when components enter states that warrant doing so. This same component would also enable routes when the state changes into something the route can handle.

A third approach would be to add a guard function as an option when the route is first registered. When the route is matched, it runs through this function, and if it passes the guard, then it is activated normally, otherwise, it fails. This approach scales best because the state that's checked; is tightly coupled with the relevant route, and there's no need to toggle between enabled/disabled states for routes. Think of a guard function as part of the matching criteria for routes.

Following is an example that shows a router that accepts guard condition functions. Route events aren't triggered if this guard function exists and returns false:

```javascript
// router.js
import events from 'events.js';

// The router triggers events in response to
// route changes.
export default class Router {
    constructor() {
        this.routes = [];
    }

    // Adds a new route, with an optional
    // guard function.
    add(pattern, name, guard) {
        this.routes.push({
            pattern: new RegExp('^' + pattern: new RegExp('(^') +
```
pattern.replace(/:\w+/g, '(.*')
name: name,
guard: guard
});

start() {
  var onHashChange = () => {
    for (let route of this.routes) {
      let guard = route.guard;
      let result = route.pattern.exec(
        location.hash.substr(1));

      // If a match is found, and there's a guard
      // condition, evaluate it. The event is only
      // triggered if this passes.
      if (result) {
        if (typeof guard === 'function' && guard()) {
          events.trigger('route:' + route.name, {
            values: result.splice(1)
          });
        }
        break;
      }
    }
    window.addEventListener('hashchange', onHashChange);
    onHashChange();
  };
}

// main.js
import Router from 'router.js';
import events from 'events.js';

var router = new Router()

  // Function that can be used as a guard condition
  // with any route we declare. It's returning a random
  // value to demonstrate the various outcomes, but this
// could be anything that we want applied to all our routes.
function isAuthorized() {
    return !!Math.round(Math.random());
}

// The first route doesn't have a guard condition,
// and will always trigger a route event. The second
// route will only trigger a route event if the given
// callback function returns true.
router.add('open', 'open');
router.add('guarded', 'guarded', isAuthorized);

events.listen('route:open', () => {
    console.log('open route is always accessible');
});

events.listen('route:guarded', (data) => {
    console.log('made it past the guard function!');
});

router.start();

Troubleshooting routers
Once our routers grow to a sufficiently large size, we'll have to troubleshoot complex
scenarios. If we know what the likely issues are beforehand, we'll be better equipped
to deal with them. We can also build troubleshooting tools into our router instances
to aid in the process. Scaling the addressability of our architecture means responding
to issues quickly and predictably.

Conflicting routes
Conflicting routes can cause a massive headache because they can be really tricky
to track down. A conflicting pattern is a general or similar version of more specific
patterns added to the router later on. The more general pattern conflicts, because
it's matched against the most specific URIs, which should have been matched by
the more specific patterns. However, they're never tested because the general route
is executed first.
When this happens, it may not be apparent at all that there's an issue with the routing because the incorrect route handler will run perfectly fine, and in the UI, everything will seem normal—except for one thing that's slightly off. If routes are processed in FIFO order, specificity matters. That is, if the more general route patterns are added first, then they'll always match against the more specific URI strings, as they're activated.

The challenge with ordering URIs like this when there's lots of them, is that it's time-consuming work. We have to compare the ordering of any new routes we may add to the patterns of existing routes. There's also the potential for conflicts between developer commitments if they're all being added to the same place. This is another advantage of separating routes by component. It makes potentially conflicting routes a lot easier to spot and deal with, because the component likely has a small number of similar URI patterns.

Following is an example that shows a router component with two conflicting routes:

```javascript
// Finds the first matching route in "routes" - tested
// against "uri".
function match() {
    for (let route of routes) {
        if (route.route.test(uri)) {
            console.log('match', route.name);
            break;
        }
    }
}

var uri = 'users/abc';
var routes = [
    { route: /^users/, name: 'users' },
    { route: /^users\/(\w+)/, name: 'user' }
];

match();
//    match users
// Note that this probably isn't expected behavior
// if we look closely at the "uri". This illustrates
// the importance of order, when testing against a
// collection of URIs specs.

routes.reverse();

match();
//    match user
Logging initial configuration

Routers shouldn't start listening to URI change events until they're configured with all the relevant routes. For example, if individual components configure the router with the routes required by that component, we wouldn't want the router to start listening for URI change events until the component has a chance to configure its routes.

The main application component that initializes its subordinate components would probably bootstrap this process, and when completed, tell the router to start. When individual components have their own routes encapsulated within, it can be difficult, during development, to grasp the router configuration in its entirety. For this, we need an option in our router that will log its entire configuration—the patterns, and the events they trigger. This helps us scale because we don't have to sacrifice modular routes to get the big picture.

Logging route events

In addition to logging the initial route configuration, it's helpful if the router can log the lifecycle that takes place when a URI change event is triggered. This is different from the event mechanism logging that we discussed in the preceding chapter—these events will log after the router triggers a route event.

If we're building a large-scale JavaScript architecture with lots of routes, we'll want to know everything about our router, and how it behaves at runtime. The router is so fundamental to the scalability of our application that we'll want to invest in the minute details here.

For example, it can be useful to get an idea of what the router is doing as it's walking through the available routes, looking for a match. It's also useful to see the result of what's parsed out of the URI string by the router, so that we can compare that to what's seen by the route event handlers downstream. Not all router components will support this level of logging. If it turns out that we need it, some frameworks will provide sufficient entry points into their components, along with good extension mechanisms.
Handling invalid resource states

Sometimes, we forget that the router is stateless; it takes a URI string as input, and triggers events based on pattern-matching criteria. A scaling problem related to addressability isn't with the router state, but the state of components that listen to routes.

For example, imagine we navigate away from one resource to another. While we're visiting this new resource, a lot can happen with that first resource. Well, it's easy for it to change in ways that make it illegal for this particular user to visit, meanwhile, it's in their history and all they need to do is hit the back button.

It's edge cases like these that routers and addressability can introduce into our application. It's not, however, the responsibility of the router to handle these edge cases. They happen due to a combination of lots of URIs, lots of components, and complex business rules that tie them all together. The router is just a mechanism to help us cope with large-scale policies, not a place to implement policies.

Summary

This chapter went into detail on addressability, and how to achieve this architectural property as our application scales.

We began our discussion of routing and addressability with a look at the different approaches to routing—the hash change event and utilizing the history API available in modern browsers. Most frameworks abstract the differences away for us. Next, we looked at the responsibilities of routers, and how they should be decoupled from other components through triggering events.

The design of URIs themselves also plays a role in the scalability of our software, because they need to be consistent and predictable. Even the users can use this predictability to help themselves scale the use of our software. URIs encode information which is then relayed to our handlers that respond to routes; this also needs to be taken into consideration.

We then looked at the various ways in which routes are triggered. The standard approach here is to click a link. If our application is well connected, it's going to have links all over the place. To help us scale lots of links, we need a way to generate URI strings automatically. Next, we're going to look at the metadata our components need in order to function. These are the user preferences and default values for our components.
Where to buy this book
You can buy JavaScript at Scale from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.