COEN 168/268, Winter 2014

Mobile Web Application Development

Ember Routing

Peter Bergström (pbergstrom@scu.edu)

Santa Clara University

Introduction to Routing

What Are Routes?

- As users interact with your application, it moves through many different states.
- In Ember.js, each of the possible states in your application is represented by a URL.
- Because app state like: *Are we logged in? What post are we looking at?* —are encapsulated by route handlers for the URLs, answering them is both simple and accurate.
- Ember.js gives you helpful tools for managing that state in a way that scales with your application.

Route Handlers

At any given time, your application has one or more *active route handlers*. The active handlers can change for one of two reasons:

1. The user interacted with a view, which generated an event that caused the URL to change.
2. The user changed the URL manually (e.g., via the back button), or the page was loaded for the first time.

Route Handlers, Cont'd

When the current URL changes, the newly active route handlers may do one or more of the following:

1. Conditionally redirect to a new URL.
2. Update a controller so that it represents a particular model.
3. Change the template on screen, or place a new template into an existing outlet.

Logging Route Changes

- As your application increases in complexity, it can be helpful to see exactly what is going on with the router.
- To have Ember write out transition events to the log, simply modify your `Ember.Application`:

```
App = Ember.Application.create({
  LOG_TRANSITIONS: true
});
```

Specifying a Root URL

- If your Ember app is one of multiple web applications served from the same domain.
- You may need to indicate to the router what the root URL for your Ember application is.
- By default, Ember assumes it is served from the domain's root.

```
App.Router.reopen({
  rootURL: '/blog/'
});
```

Defining Your Routes

Defining Your Routes

- When your app starts, the router is responsible for displaying templates, loading data, and otherwise setting up app state.
- It does so by matching the current URL to the defined *routes*

```
App.Router.map(function() {
  this.route("about", { path: "/about" });
  this.route("favorites", { path: "/favs" });
});
```

Now, when the user visits `/about`, Ember.js will render the `about` template. Visiting `/favs` will render the `favorites` template.

You Don't Have Specify All Routes!

You get a few routes for free:
- The `ApplicationRoute`
- The `IndexRoute` (corresponding to the `/` path).

Also...

You can leave off the path if it is the same as the route name. In this case, the following is equivalent to the above example:

```
App.Router.map(function() {
  this.route("about");
  this.route("favorites", { path: "/favs" });
});
```

Linking To Different Routes

- Inside your templates, you can use `{{link-to}}` to navigate between routes.
- Use the name that you provided to the `route` method (or, in the case of /, the name `index`).

```
{{#link-to 'index'}}<img class="logo">{{/link-to}}

<nav>
  {{#link-to 'about'}}About{{/link-to}}
  {{#link-to 'favorites'}}Favorites{{/link-to}}
</nav>
```

Customizing The Behavior Of A Route

- Creating an `Ember.Route` subclass:

```
App.IndexRoute = Ember.Route.extend({
  setupController: function(controller) {
    // Set the IndexController's `title`
    controller.set('title', "My App");
  }
});
```

The `IndexController` is the starting context for the `index` template.

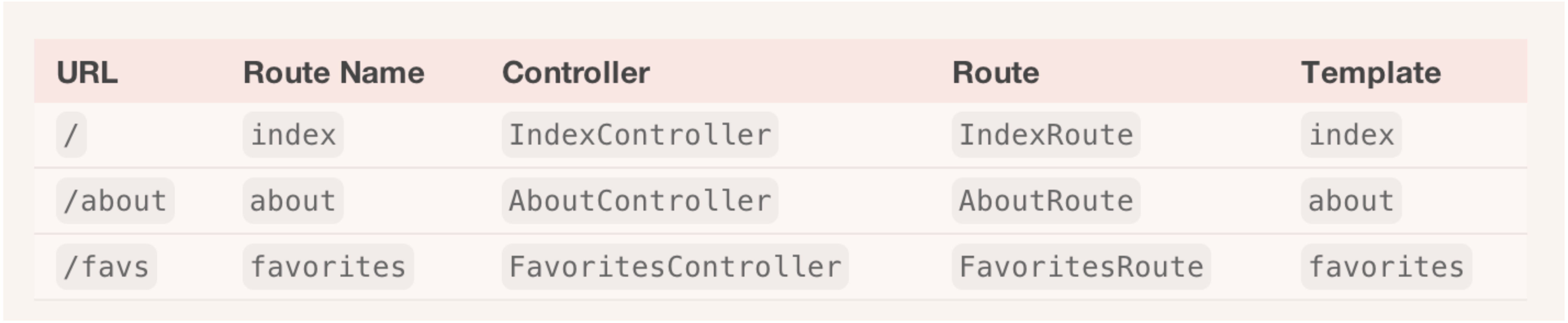
Customizing The Behavior Of A Route, Cont'd

Now that you've set `title`, you can use it in the template:

```
<!-- get the title from the IndexController -->
<h1>{{title}}</h1>
```

(If you don't explicitly define an `App.IndexController`, Ember.js will automatically generate one for you.)

Ember.js automatically figures out the names of the routes and controllers based on the name you pass to `this.route`.



URL	Route Name	Controller	Route	Template
`/`	`index`	`IndexController`	`IndexRoute`	`index`
`/about`	`about`	`AboutController`	`AboutRoute`	`about`
`/favs`	`favorites`	`FavoritesController`	`FavoritesRoute`	`favorites`

Resources

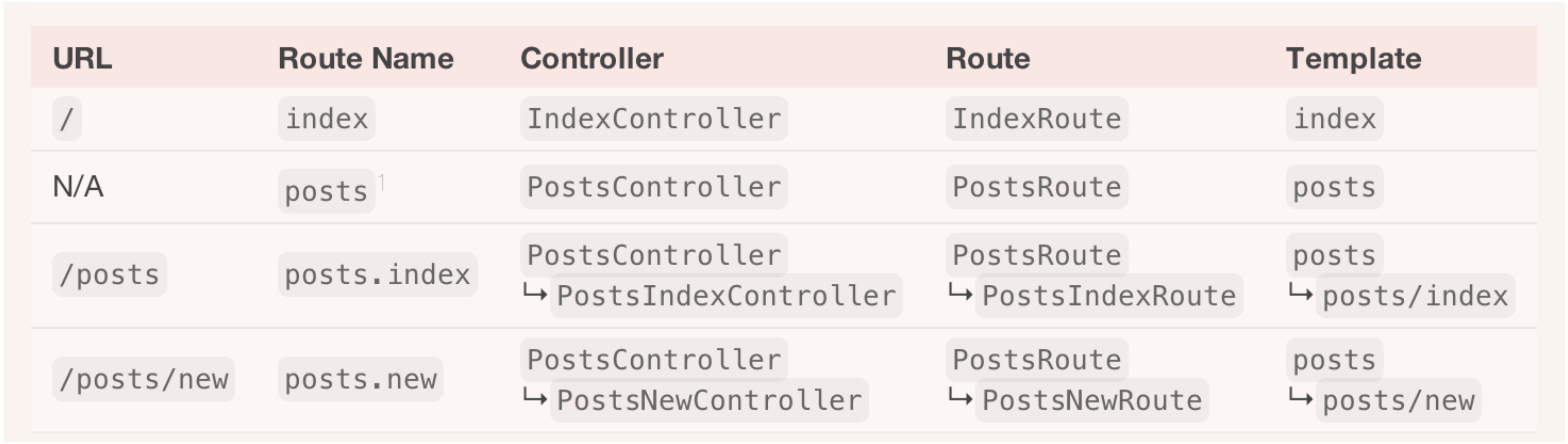
You can define groups of routes that work with a resource:

```
App.Router.map(function() {
  this.resource('posts', { path: '/posts' }, function() {
    this.route('new');
  });
});
```

As with `this.route`, you can leave off the path if it's the same as the name of the route, so the following router is equivalent:

```
App.Router.map(function() {
  this.resource('posts', function() {
    this.route('new');
  });
});
```

This router creates three routes:



URL	Route Name	Controller	Route	Template
`/`	`index`	`IndexController`	`IndexRoute`	`index`
N/A	`posts`[1]	`PostsController`	`PostsRoute`	`posts`
`/posts`	`posts.index`	`PostsController` ↳ `PostsIndexController`	`PostsRoute` ↳ `PostsIndexRoute`	`posts` ↳ `posts/index`
`/posts/new`	`posts.new`	`PostsController` ↳ `PostsNewController`	`PostsRoute` ↳ `PostsNewRoute`	`posts` ↳ `posts/new`

Resources vs Routes

- A `resource` should be used for URLs that represent a **noun**
- A `route` should be used for URLs that represent **adjectives** or **verbs**

For example, when specifying URLs for posts (a noun), the route was defined with `this.resource('posts')`. However, when defining the new action (a verb), the route was defined with `this.route('new')`.

Routes Are Used To Convert URLS to models

For example, if we have the resource `this.resource('posts');`, our route handler might look like this:

```
App.PostsRoute = Ember.Route.extend({
  model: function() {
    return this.store.find('posts');
  }
});
```

The `posts` template will then receive a list of all available posts as its context.

Dynamic Routes Are Needed

- `/posts` represents a fixed model, we don't need any additional information to know what to retrieve.
- However, if we want a route to represent a single post, we would not want to have to hardcode every possible post into the router.

Therefore, we need *dynamic segments*

What is a Dynamic Segment?

A dynamic segment is a portion of a URL that starts with a `:` and is followed by an identifier.

```
App.Router.map(function() {
  this.resource('posts');
  this.resource('post', { path: '/post/:post_id' });
});

App.PostRoute = Ember.Route.extend({
  model: function(params) {
    return this.store.find('post', params.post_id);
  }
});
```

This pattern is so common, the `model` hook is the default behavior

- If the dynamic segment is `:post_id`, Ember.js is smart enough to know that it should use the model `App.Post` (with the ID provided in the URL).
- Specifically, unless you override `model`, the route will return `this.store.find('post', params.post_id)` automatically.

What About If Your Model Does Not Use `id`?

If your model does not use the `id` property in the URL, you should define a serialize method on your route:

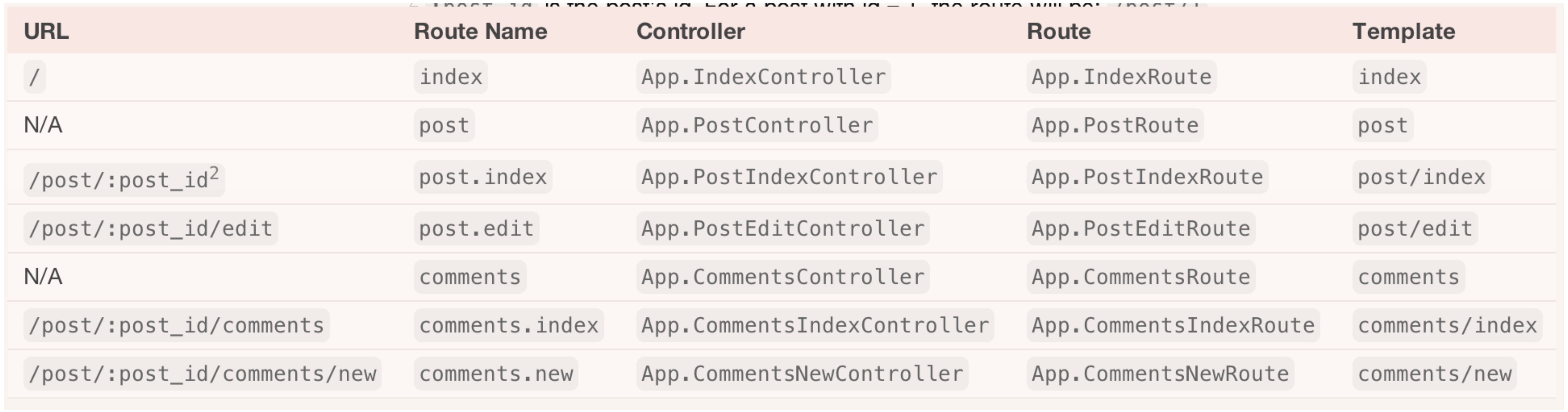
```
App.Router.map(function() {
  this.resource('post', {path: '/posts/:post_slug'});
});

App.PostRoute = Ember.Route.extend({
  model: function(params) {
    // the server returns `{ slug: 'foo-post' }`
    return jQuery.getJSON("/posts/" + params.post_slug);
  },

  serialize: function(model) { // default inserts models `id` into the route
    // this will make the URL `/posts/foo-post`
    return { post_slug: model.get('slug') };
  }
});
```

Nested Resources

You cannot nest routes, but you can nest resources:

```
App.Router.map(function() {
  this.resource('post', { path: '/post/:post_id' }, function() {
    this.route('edit');
    this.resource('comments', function() {
      this.route('new');
    });
  });
});
```

Nested Resources, Continued



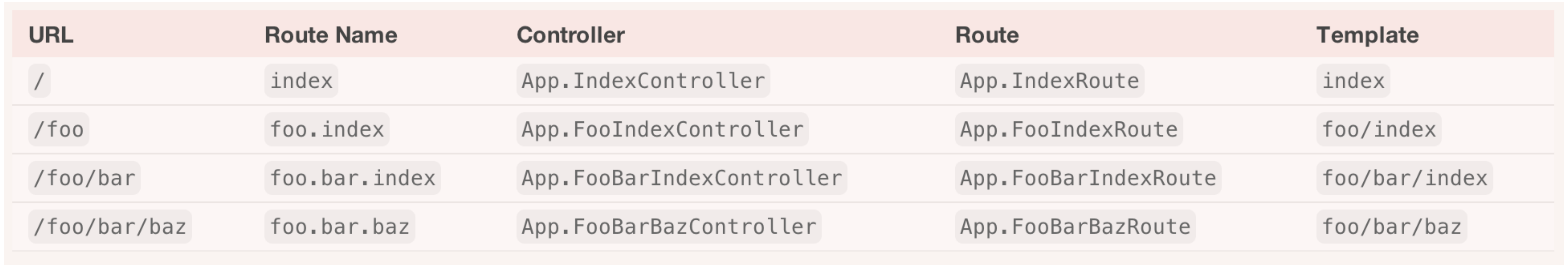
URL	Route Name	Controller	Route	Template
`/`	`index`	`App.IndexController`	`App.IndexRoute`	`index`
N/A	`post`	`App.PostController`	`App.PostRoute`	`post`
`/post/:post_id`[2]	`post.index`	`App.PostIndexController`	`App.PostIndexRoute`	`post/index`
`/post/:post_id/edit`	`post.edit`	`App.PostEditController`	`App.PostEditRoute`	`post/edit`
N/A	`comments`	`App.CommentsController`	`App.CommentsRoute`	`comments`
`/post/:post_id/comments`	`comments.index`	`App.CommentsIndexController`	`App.CommentsIndexRoute`	`comments/index`
`/post/:post_id/comments/new`	`comments.new`	`App.CommentsNewController`	`App.CommentsNewRoute`	`comments/new`

Creating Deeply Nested Resources

You are also able to create deeply nested resources in order to preserve the namespace on your routes:

```
App.Router.map(function() {
  this.resource('foo', function() {
    this.resource('foo.bar', { path: '/bar' }, function() {
      this.route('baz'); // This will be foo.bar.baz
    });
  });
});
```

Creating Deeply Nested Resources, Cont'd



URL	Route Name	Controller	Route	Template
`/`	`index`	`App.IndexController`	`App.IndexRoute`	`index`
`/foo`	`foo.index`	`App.FooIndexController`	`App.FooIndexRoute`	`foo/index`
`/foo/bar`	`foo.bar.index`	`App.FooBarIndexController`	`App.FooBarIndexRoute`	`foo/bar/index`
`/foo/bar/baz`	`foo.bar.baz`	`App.FooBarBazController`	`App.FooBarBazRoute`	`foo/bar/baz`

Initial routes

A few routes are immediately available within your application:

- `App.ApplicationRoute` is entered when your app first boots up. It renders the `application` template.

- `App.IndexRoute` is the default route, and will render the `index` template when the user visits `/` (unless `/` has been overridden by your own custom route).

Remember, these routes are part of every application, so you don't need to specify them in `App.Router.map`.

Wildcard / globbing Routes

- You can define wildcard routes that will match multiple routes.
- This could be used if you'd like a catchall route which is useful when the user enters an incorrect URL not managed by your app.

Wildcard / globbing Routes Example

```
App.Router.map(function() {
  this.route('catchall', {path: '/*wildcard'});
});
```

Like all routes with a dynamic segment, you must provide a context when using a `{{link-to}}` or `transitionTo` to programatically enter this route.

Wildcard / globbing Routes Example cont'd

```
App.ApplicationRoute = Ember.Route.extend({
  actions: {
    error: function () {
      this.transitionTo('catchall', "application-error");
    }
  }
});
```

With this code, if an error bubbles up to the Application route, your application will enter the `catchall` route and display / `application-error` in the URL.

Generated Objects

Generated Objects

- Whenever you define a new route, Ember.js attempts to find corresponding Route, Controller, View, and Template classes named according to naming conventions.
- If an implementation of any of these objects is not found, appropriate objects will be generated in memory for you.

Generated Routes

Given you have the following route:

```
App.Router.map(function() {
  this.resource('posts');
});
```

- When you navigate to `/posts`, Ember.js looks for `App.PostsRoute`.
- If it doesn't find it, it will automatically generate an `App.PostsRoute` for you.

Custom Generated Routes

You can have all your generated routes extend a custom route. If you define `App.Route`, all generated routes will be instances of that route.

Generated Controllers

- If you navigate to route `posts`, Ember.js looks for a controller called `App.PostsController`.
- If you did not define it, one will be generated for you.
- Ember.js can generate three types of controllers:
  - `Ember.ObjectController`, `Ember.ArrayController`, and `Ember.Controller`.

The Type of Generated Controller Depends On The `model` Hook

- If it returns an object (such as a single record), an `ObjectController` will be generated.
- If it returns an array, an `ArrayController` will be generated.
- If it does not return anything, an instance of `Ember.Controller` will be generated.

Custom Generated Controllers

If you want to customize generated controllers, you can define your own:
- `App.Controller`
- `App.ObjectController`
- `App.ArrayController`.

Generated controllers will extend one of these three (depending on the conditions above).

Generated Views and Templates

- A route also expects a view and a template.
- If you don't define a view, a view will be generated for you.
- A generated template is empty.
- If it's a resource template, the template will simply act as an `outlet` so that nested routes can be seamlessly inserted.

```
// It is equivalent to:
{{outlet}}
```

Specifying A Route's Model

Templates Are Backed By Models

- How do templates know which model they should display?
- This is one of the jobs of an `Ember.Route`.
- You can tell a template which model it should render by defining a route with the same name as the template, and implementing its `model` hook.

For example, to provide some model data to the `photos` template, we would define an `App.PhotosRoute` object:

```
App.PhotosRoute = Ember.Route.extend({
  model: function() {
    return [{
      title: "Tomster",
      url: "http://emberjs.com/images/about/ember-productivity-sm.png"
    }, {
      title: "Eiffel Tower",
      url: "http://emberjs.com/images/about/ember-structure-sm.png"
    }];
  }
});
```

Asynchronously Loading Models

- In the above example, the model data was returned synchronously from the `model` hook.
- This means that the data was available immediately and your application did not need to wait for it to load, in this case because we immediately returned an array of hardcoded data.
- Of course, this is not always realistic.
- Usually, the data will not be available synchronously, but instead must be loaded asynchronously over the network.

Using "promises" to manage data loading

- In cases where data is available asynchronously, you can just return a promise from the `model` hook, and Ember will wait until that promise is resolved before rendering the template.
- The basic idea is that they are objects that represent eventual values.
- Ex: If you use jQuery's `getJSON()` method, it will return a promise for the JSON that is eventually returned.
- Ember uses this promise object to know when it has enough data to continue rendering.

Here's a route that loads the most recent PRs sent to Ember.js:

```
App.PullRequestsRoute = Ember.Route.extend({
  model: function() {
    return Ember.$.getJSON('https://api.github.com/repos/emberjs/ember.js/pulls');
  }
});
```

- Looks like it's synchronous, making it easy to read and reason about, it's actually completely asynchronous.
- Ember detects that a promise is returned from the `model` hook, and wait until that promise resolves to render the `pullRequests` template

Benefits of Promises

- Because Ember supports promises, it can work with any persistence library that uses them as part of its public API.
- You can also use many of the conveniences built in to promises to make your code even nicer.
- For example, imagine if we wanted to modify the above example so that the template only displayed the three most recent pull requests.

We can rely on promise chaining to modify the data returned from the JSON request before it gets passed to the template:

```
App.PullRequestsRoute = Ember.Route.extend({
  model: function() {
    var url = 'https://api.github.com/repos/emberjs/ember.js/pulls';
    return Ember.$.getJSON(url).then(function(data) {
      return data.splice(0, 3);
    });
  }
});
```

Setting Up Controllers with the Model

- So what actually happens with the value you return from the `model` hook?
- By default, the value returned from your `model` hook will be assigned to the `model` property of the associated controller. For example, if your `App.PostsRoute` returns an object from its `model` hook, that object will be set as the `model` property of the `App.PostsController`.

(This, under the hood, is how templates know which model to render: they look at their associated controller's `model` property.)

Dynamic Models

- Some routes always display the same model.
- For example, the `/photos` route will always display the same list of photos available in the application.
- If your user leaves this route and comes back later, the model does not change.
- However, you will often have a route whose model will change depending on user interaction.

Imagine a photo viewer app

- The `/photos` route will render the `photos` template with the list of photos as the model, which never changes.
- But when the user clicks on a particular photo, we want to display that model with the `photo` template. If the user goes back and clicks on a different photo, we want to display the `photo` template again, this time with a different model.
- In cases like this, it's important that we include some information in the URL about not only which template to display, but also which model.

In Ember, define routes with *dynamic segments*.

- A dynamic segment is a part of the URL that is filled in by the current model's ID.
- Dynamic segments always start with a colon (`:`).
- Our photo example might have its `photo` route defined like this:

```
App.Router.map(function() {
  this.resource('photo', { path: '/photos/:photo_id' });
});
```

Dynamic Segments for Photo Route

- The `photo` route has a dynamic segment `:photo_id`.
- When the user goes to the `photo` route to display a particular photo model (usually via the `{{link-to}}` helper), that model's ID will be placed into the URL automatically.
- For example, if you transitioned to the `photo` route with a model whose `id` property was 47, the URL in the user's browser would be updated to:

```
/photos/47
```

Going Directly To A URL with a Dynamic Segment

- Users might reload the page, or send the link to a friend, who clicks on it.
- At that point, because we are starting the application up from scratch, the actual JavaScript model object to display has been lost; all we have is the ID from the URL.
- Luckily, Ember will extract any dynamic segments from the URL for you and pass them as a hash to the `model` hook as the first argument.

You Always Need to Load The `id` In the Route

```
App.Router.map(function() {
  this.resource('photo', { path: '/photos/:photo_id' });
});

App.PhotoRoute = Ember.Route.extend({
  model: function(params) {
    return Ember.$.getJSON('/photos/'+params.photo_id);
  }
});
```

In the above example, we construct a URL for the JSON representation of that photo. Once we have the URL, we use jQuery to return a promise for the JSON model data.

Ember Data

- Many Ember developers use a model library to make finding and saving records easier than manually managing Ajax calls.
- In particular, using a model library allows you to cache records that have been loaded, significantly improving the performance of your application.
- One popular model library built for Ember is Ember Data. We will talk about it later.

Setting Up A Controller

- Changing the URL may also change which template is displayed on screen.
- Templates, however, are usually only useful if they have some source of information to display.
- In Ember.js, a template retrieves information to display from a controller.

There Are Two Types of Controllers

- `Ember.ObjectController`
  - Displays one model object
- `Ember.ArrayController`
  - Displays an array of model objects

How do you specify the controller?

Set its `model` property in the route handler's `setupController` hook.

```
App.Router.map(function() {
  this.resource('post', { path: '/posts/:post_id' });
});

App.PostRoute = Ember.Route.extend({
  // The code below is the default behavior, so if this is all you
  // need, you do not need to provide a setupController implementation
  // at all.
  setupController: function(controller, model) {
    controller.set('model', model);
  }
});
```

What does `setupController` do?

- Receives the route handler's associated controller as its first argument.
- In this case, the `PostRoute`'s `setupController` receives the application's instance of `App.PostController`.
- As a second argument, it receives the route handler's model.

To specify a controller other than the default, set the route's `controllerName` property:

```
App.SpecialPostRoute = Ember.Route.extend({
  controllerName: 'post'
});
```

To configure a controller other than the controller associated with the route handler, use the `controllerFor` method:

```
App.PostRoute = Ember.Route.extend({
  setupController: function(controller, model) {
    this.controllerFor('topPost').set('model', model);
  }
});
```

Rendering A Template

Rendering A Template

- One of the most important jobs of a route handler is rendering the appropriate template to the screen.
- By default, a route handler will render the template into the closest parent with a template.

```
App.Router.map(function() {
  this.resource('posts');
});

App.PostsRoute = Ember.Route.extend();
```

If you want to render a template other than the one associated with the route handler, implement the `renderTemplate` hook:

```
App.PostsRoute = Ember.Route.extend({
  renderTemplate: function() {
    this.render('favoritePost');
  }
});
```

If you want to use a different controller than the route handler's controller, pass the controller's name in the `controller` option:

```
App.PostsRoute = Ember.Route.extend({
  renderTemplate: function() {
    this.render({ controller: 'favoritePost' });
  }
});
```

Ember allows you to name your outlets. For instance, this code allows you to specify two outlets with distinct names:

```
<div class="toolbar">{{outlet toolbar}}</div>
<div class="sidebar">{{outlet sidebar}}</div>
```

So, if you want to render your posts into the `sidebar` outlet, use code like this:

```
App.PostsRoute = Ember.Route.extend({
  renderTemplate: function() {
    this.render({ outlet: 'sidebar' });
  }
});
```

All of the options described above can be used together in whatever combination you'd like

```
App.PostsRoute = Ember.Route.extend({
  renderTemplate: function() {
    var controller = this.controllerFor('favoritePost');

    // Render the `favoritePost` template into
    // the outlet `posts`, and display the `favoritePost`
    // controller.
    this.render('favoritePost', {
      outlet: 'posts',
      controller: controller
    });
   }
});
```

If you want to render two different templates into outlets of two different rendered templates of a route:

```
App.PostRoute = App.Route.extend({
  renderTemplate: function() {
    this.render('favoritePost', {   // the template to render
      into: 'posts',                // the template to render into
      outlet: 'posts',              // the name of the outlet in that template
      controller: 'blogPost'        // the controller to use for the template
    });
    this.render('comments', {
      into: 'favoritePost',
      outlet: 'comment',
      controller: 'blogPost'
    });
   }
});
```

Transitioning and Redirecting

Using `transitionTo` or `transitionToRoute`

Calling `transitionTo` from a route or `transitionToRoute` from a controller will stop any transition currently in progress and start a new one, functioning as a redirect.

`transitionTo` takes params and behaves like `{{link-to}}` helper

- If you transition into a route without dynamic segments that route's `model` hook will always run.
- If the new route has dynamic segments, you need to pass either a *model* or an *identifier* for each segment.
- Passing a model will skip that segment's `model` hook.
- Passing an identifier will run the `model` hook and you'll be able to access the identifier in the params

Before the model is known

If you want to redirect from one route to another, you can do the transition in the `beforeModel` hook of your route handler.

```
App.Router.map(function() {
  this.resource('posts');
});

App.IndexRoute = Ember.Route.extend({
  beforeModel: function() {
    this.transitionTo('posts');
  }
});
```

After the model is known

- If you need some information about the current model in order to decide about the redirection, you should either use the `afterModel` or the `redirect` hook.
- They receive the resolved model as the first parameter and the transition as the second one, and thus function as aliases.
- In fact, the default implementation of `afterModel` just calls `redirect`.

```
App.Router.map(function() {
  this.resource('posts');
  this.resource('post', { path: '/post/:post_id' });
});

App.PostsRoute = Ember.Route.extend({
  afterModel: function(posts, transition) {
    if (posts.get('length') === 1) {
      this.transitionTo('post', posts.get('firstObject'));
    }
  }
});
```

When transitioning to the `PostsRoute` if it turns out that there is only one post, the current transition will be aborted in favor of redirecting to the `PostRoute` with the single post object being its model.

Based on other application state

You can conditionally transition based on some other application state.

(see next slide)

```
App.Router.map(function() {
  this.resource('topCharts', function() {
    this.route('choose', { path: '/' });
    this.route('albums');
    this.route('songs');
    this.route('artists');
    this.route('playlists');
  });
});

App.TopChartsChooseRoute = Ember.Route.extend({
  beforeModel: function() {
    var lastFilter = this.controllerFor('application').get('lastFilter');
    this.transitionTo('topCharts.' + (lastFilter || 'songs'));
  }
});

// Superclass to be used by all of the filter routes below
App.FilterRoute = Ember.Route.extend({
  activate: function() {
    var controller = this.controllerFor('application');
    controller.set('lastFilter', this.templateName);
  }
});

App.TopChartsSongsRoute = App.FilterRoute.extend();
App.TopChartsAlbumsRoute = App.FilterRoute.extend();
App.TopChartsArtistsRoute = App.FilterRoute.extend();
App.TopChartsPlaylistsRoute = App.FilterRoute.extend();
```

Sorry, that was a lot of code!

- In this example, navigating to the `/` URL immediately transitions into the last filter URL that the user was at.
- The first time, it transitions to the `/songs` URL.
- Your route can also choose to transition only in some cases.
- If the `beforeModel` hook does not abort or transition to a new route, the remaining hooks (`model`, `afterModel`, `setupController`, `renderTemplate`) will execute as usual.

Specifying The URL Type

Specifying The URL Type

- By default the Router uses the browser's hash to load the starting state of your application and will keep it in sync as you move around.
- At present, this relies on a `hashchange` event existing in the browser.

Default Behavior

Given the following router, entering `/#/posts/new` will take you to the `posts.new` route.

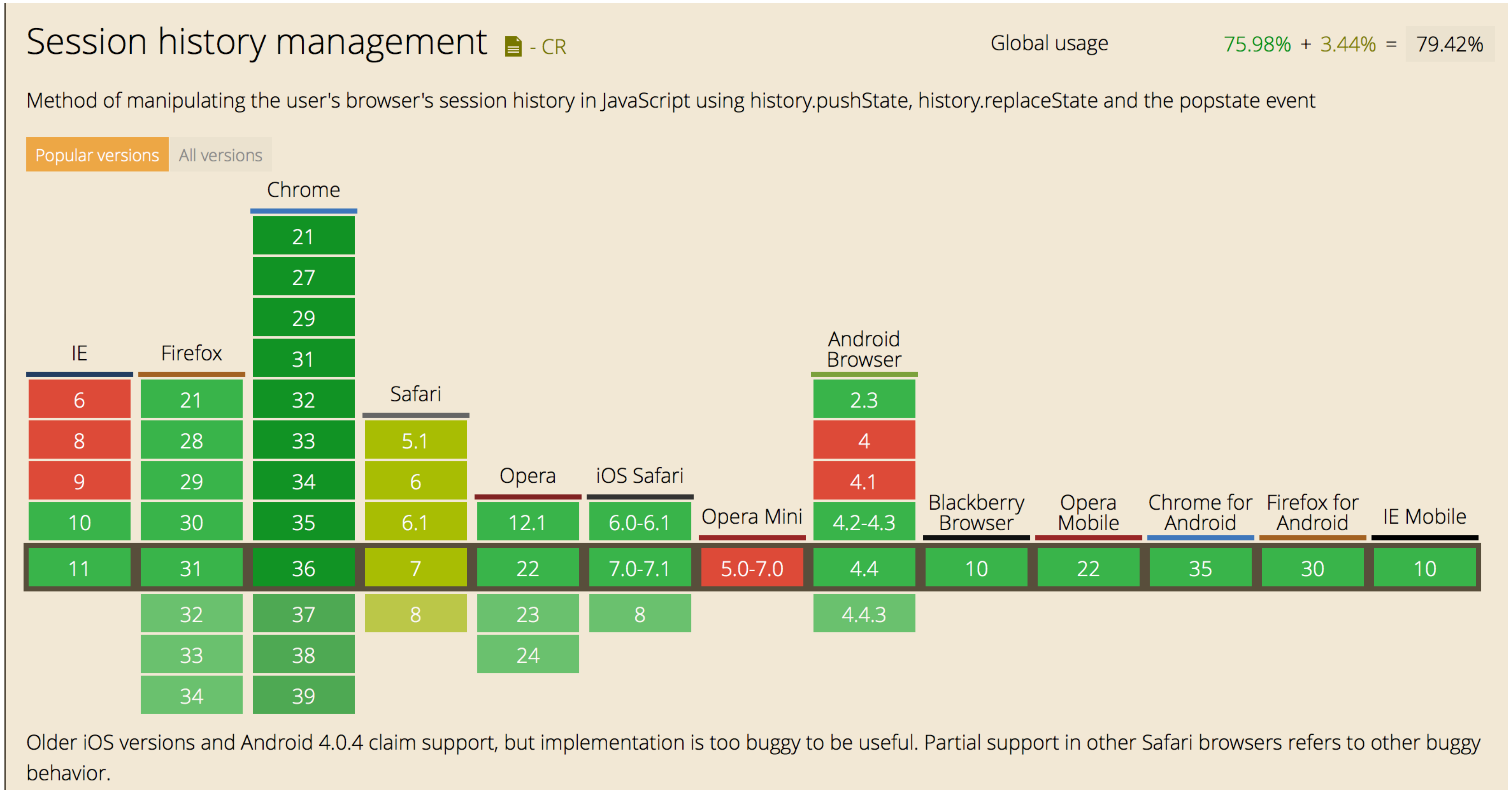
```
App.Router.map(function() {
  this.resource('posts', function() {
    this.route('new');
  });
});
```

If you want to use a regular URL path...

If you want `/posts/new` to work instead, you can tell the Router to use the browser's history API.

Keep in mind that your server must serve the Ember app at all the routes defined here.

```
App.Router.reopen({
  location: 'history'
});
```



Session history management - CR

Global usage 75.98% + 3.44% = 79.42%

Method of manipulating the user's browser's session history in JavaScript using history.pushState, history.replaceState and the popstate event

Popular versions | All versions

IE	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Android Browser	Blackberry Browser	Opera Mobile	Chrome for Android	Firefox for Android	IE Mobile
		21										
		27										
		29										
		31										
6	21	32					2.3					
8	28	33	5.1				4					
9	29	34	6				4.1					
10	30	35	6.1	12.1	6.0-6.1		4.2-4.3					
11	31	36	7	22	7.0-7.1	5.0-7.0	4.4	10	22	35	30	10
	32	37	8	23	8		4.4.3					
	33	38		24								
	34	39										

Older iOS versions and Android 4.0.4 claim support, but implementation is too buggy to be useful. Partial support in other Safari browsers refers to other buggy behavior.

Source: http://caniuse.com/history

But, what if you don't want URLs at all?

- Finally, if you don't want the browser's URL to interact with your application at all, you can disable the location API entirely.
- This is useful for testing, or when you need to manage state with your Router, but temporarily don't want it to muck with the URL (for example when you embed your application in a larger page).

```
App.Router.reopen({
  location: 'none'
});
```

Query Parameters

Query Parameters

- Query parameters are optional key-value pairs that appear to the right of the ? in a URL.
- For example, the following URL has two query params, `sort` and `page`, which respective values ASC and 2:

```
http://example.com/articles?sort=ASC&page=2
```

- Query params allow for additional application state to be serialized into the URL that can't otherwise fit into the *path* of the URL (i.e. everything to the left of the ?).

Specifying Query Parameters

- Query params can be declared on route-driven controllers
  - e.g. to configure query params that are active within the `articles` route, they must be declared on `ArticlesController`.
- **Note:** The controller associated with a given route can be changed by specifying the `controllerName` property on that route.

A Query Parameter Example

- Let's say we'd like to add a `category` query parameter that will filter out all the articles that haven't been categorized as popular.
- To do this, we specify `'category'` as one of `ArticlesController`'s `queryParams`:

```
App.ArticlesController = Ember.ArrayController.extend({
  queryParams: ['category'],
  category: null
});
```

A Query Parameter Example, Cont'd

- This sets up a binding between the `category` query param in the URL, and the `category` property on `ArticlesController`.
- Once the `articles` route has been entered, any changes to the `category` query param in the URL will update the `category` property on `ArticlesController`, and vice versa.

Now we just need to define a computed property of our category-filtered array that `articles` template will render:

```
App.ArticlesController = Ember.ArrayController.extend({
  queryParams: ['category'],
  category: null,

  filteredArticles: function() {
    var category = this.get('category');
    var articles = this.get('model');

    if (category) {
      return articles.filterBy('category', category);
    } else {
      return articles;
    }
  }.property('category', 'model')
});
```

With this code, we have established the following behaviors:

1. If the user navigates to `/articles`, `category` will be `null`, so the articles won't be filtered.
2. If the user navigates to `/articles?articles[category]=recent`, `category` will be set to `"recent"`, so articles will be filtered.
3. Once inside the `articles` route, any changes to the `category` property on `ArticlesController` will cause the URL to update the query param.

`{{link-to}}` Helper

The `link-to` helper supports specifying query params by way of the `query-params` subexpression helper.

```
// Explicitly set target query params
{{#link-to 'posts' (query-params direction="asc")}}Sort{{/link-to}}

// Binding is also supported
{{#link-to 'posts' (query-params direction=otherDirection)}}Sort{{/link-to}}
```

- In the previous example, `direction` is presumably a query param property on `PostsController`.
- But it could also refer to a `direction` property on any of the controllers associated with the `posts` route hierarchy, matching the leaf-most controller with the supplied property name.
- The link-to helper takes into account query parameters when determining its "active" state, and will set the class appropriately.
- You don't have to supply all of the current, active query params for this to be true.

transitionTo

Route#transitionTo (and Controller#transitionToRoute) now accepts a final argument, which is an object with the key queryParams.

```
this.transitionTo('post', object, {queryParams: {showDetails: true}});
this.transitionTo('posts', {queryParams: {sort: 'title'}});

// if you just want to transition the query parameters without changing the route
this.transitionTo({queryParams: {direction: 'asc'}});
```

You can also add query params to URL transitions:

```
this.transitionTo("/posts/1?sort=date&showDetails=true");
```

Opting into a full transition

- If the arguments in `transitionTo` or `link-to` only change query params
- It is not considered a full transition
- This means that hooks like `model` and `setupController` won't fire by default
- Only controller properties will be updated along with the URL

However, you can opt in to a full transition

- Some query param changes necessitate loading data from the server
- Then it is desirable to opt into a full-on transition
- To opt into a full transition when a controller query param property changes, you can use the optional `queryParams` configuration hash on the `Route` associated with that controller, and set that query param's `refreshModel` config property to `true`

```
App.ArticlesRoute = Ember.Route.extend({
  queryParams: {
    category: {
      refreshModel: true
    }
  },
  model: function(params) {
    // This gets called upon entering 'articles' route
    // for the first time, and we opt in refiring it
    // upon query param changes via `queryParamsDidChange` action

    // params has format of { category: "someValueOrJustNull" },
    // which we can just forward to the server.
    return this.store.findQuery('articles', params);
  }
});

App.ArticlesController = Ember.ArrayController.extend({
  queryParams: ['category'],
  category: null
});
```

Update URL with `replaceState` instead

- By default, Ember will use `pushState` to update the URL in the address bar in response to a controller query param property change
- But if you would like to use `replaceState` instead (which prevents an additional item from being added to your browser's history)
- Specify this on the `Route`'s `queryParams` config hash.

```
App.ArticlesRoute = Ember.Route.extend({
  queryParams: {
    category: {
      replace: true
    }
  }
});
```

Note that the name of this config property and its default value of `false` is similar to the `link-to` helper's, which also lets you opt into a `replaceState` transition via `replace=true`.

Map a controller's property to a different query param key

- By default, specifying `foo` as a controller query param property will bind to a query param whose key is `foo`, e.g. `?foo=123`.
- You can also map a controller property to a different query param key using the following configuration syntax:

```
App.ArticlesController = Ember.ArrayController.extend({
  queryParams: {
    category: "articles_category"
  },
  category: null
});
```

Note that query params that require additional customization can be provided along with strings in the `queryParams` array.

```
App.ArticlesController = Ember.ArrayController.extend({
  queryParams: [ "page", "filter", {
    category: "articles_category"
  }],
  category: null,
  page: 1,
  filter: "recent"
});
```

Default values and deserialization

In the following example, the controller query param property page is considered to have a default value of 1.

```
App.ArticlesController = Ember.ArrayController.extend({
  queryParams: 'page',
  page: 1
});
```

This affects query param behavior in two ways:

The type of the default value is used to cast changed query param values in the URL before setting values on the controller:

- If the user clicks the back button to change from `/?page=3` to `/?page=2`, Ember will update the `page` controller property to the properly cast number 2 rather than the string `"2"`, which it knows to do because the default value (1) is a number.

- This also allows boolean default values to be correctly cast when deserializing from URL changes.

When a controller's query param property is currently set to its default value, this value won't be serialized into the URL.

- If `page` is 1, the URL might look like `/articles`, but once someone sets the controller's `page` value to 2, the URL will become `/articles?page=2`.

Asynchronous Routing

This section covers some more advanced features of the router and its capability for handling complex async logic within your app.

A Word on Promises...

- Ember's approach to handling asynchronous logic in the router makes heavy use of the concept of Promises.
- Promises are objects that represent an eventual value.
- A promise can either *fulfill* (successfully resolve the value) or *reject* (fail to resolve the value).

A Word on Promises...

- The way to retrieve this eventual value, or handle the cases when the promise rejects, is via the promise's `then` method, which accepts two optional callbacks, one for fulfillment and one for rejection.
- If the promise fulfills, the fulfillment handler gets called with the fulfilled value as its sole argument, and if the promise rejects, the rejection handler gets called with a reason for the rejection as its sole argument.

```javascript
var promise = fetchTheAnswer();

promise.then(fulfill, reject);

function fulfill(answer) {
  console.log("The answer is " + answer);
}

function reject(reason) {
 console.log("Couldn't get the answer! Reason: " + reason);
}
```

Much of the power of promises comes from the fact that they can be chained together to perform sequential asynchronous operations:

```
// Note: jQuery AJAX methods return promises
var usernamesPromise = Ember.$.getJSON('/usernames.json');

usernamesPromise.then(fetchPhotosOfUsers)
                .then(applyInstagramFilters)
                .then(uploadTrendyPhotoAlbum)
                .then(displaySuccessMessage, handleErrors);
```

The Router Pauses for Promises

- When transitioning between routes, the router collects all of the models (via the `model` hook) that will be passed to the route's controllers at the end of the transition.
- If the `model` hook (or the related `beforeModel` or `afterModel` hooks) returns:
    - normal (non-promise) objects or arrays, the transition will complete immediately.
    - a promise (or if a promise was an arg to `transitionTo`), the it will pause until that promise fulfills or rejects.

If the promise fulfills:

- The transition will pick up where it left off
- begin resolving the next (child) route's model
- pausing if it too is a promise, and so on, until all destination route models have been resolved.

The values passed to the `setupController` hook for each route will be the fulfilled values from the promises.

A basic example:

```
App.TardyRoute = Ember.Route.extend({
  model: function() {
    return new Ember.RSVP.Promise(function(resolve) {
      Ember.run.later(function() {
        resolve({ msg: "Hold Your Horses" });
      }, 3000);
    });
  },

  setupController: function(controller, model) {
    console.log(model.msg); // "Hold Your Horses"
  }
});
```

When transitioning into `TardyRoute`:

- the `model` hook will be called
- returns a promise that won't resolve until 3 seconds later
- during which time the router will be paused in mid-transition.

Then the promise eventually fulfills, the router will continue transitioning and eventually call `TardyRoute`'s `setupController` hook with the resolved object.

This pause-on-promise behavior is extremely valuable for when you need to guarantee that a route's data has fully loaded before displaying a new template.

When Promises Reject During a Transition...

- the transition is aborted
- no new destination route templates are rendered
- an error is logged to the console.

You can configure this error-handling logic via the `error` handler on the route's `actions` hash. When a promise rejects, an `error` event will be fired on that route and bubble up to `ApplicationRoute`'s default error handler unless it is handled by a custom error handler along the way.

A Custom Error Handler…

```javascript
App.GoodForNothingRoute = Ember.Route.extend({
  model: function() {
    return Ember.RSVP.reject("FAIL");
  },

  actions: {
    error: function(reason) {
      alert(reason); // "FAIL"

      // Can transition to another route here, e.g.
      // this.transitionTo('index');

      // Uncomment the line below to bubble this error event:
      // return true;
    }
 }
});
```

Recovering from Rejection

Rejected model promises halt transitions, but because promises are chainable, you can catch promise rejects within the `model` hook itself and convert them into fulfills that won't halt the transition.

```
App.FunkyRoute = Ember.Route.extend({
  model: function() {
    return iHopeThisWorks().then(null, function() {
      // Promise rejected, fulfill with some default value to
      // use as the route's model and continue on with the transition
      return { msg: "Recovered from rejected promise" };
    });
  }
});
```

Use `beforeModel` and `afterModel` to perform any logic when:

- The `model` hook covers many use cases for pause-on-promise transitions, but sometimes you'll need `beforeModel` and `afterModel`.
- The most common reason for this is that if you're transitioning into a route with a dynamic URL segment via `{{link-to}}` or `transitionTo`
- The model for the route you're transitioning into will have already been specified which case the `model` hook won't get called

The `beforeModel` hook

- Easily the more useful of the two
- Called before the router attempts to resolve the model for the given route.

Like `model`, returning a promise from `beforeModel` will pause the transition until it resolves, or will fire an `error` if it rejects.

Some `beforeModel` use cases

- Deciding whether to redirect to another route before performing a potentially wasteful server query in `model`
- Ensuring that the user has an authentication token before proceeding onward to `model`
- Loading application code required by this route

Example `beforeModel` usage

```
App.SecretArticlesRoute  = Ember.Route.extend({
  beforeModel: function() {
    if (!this.controllerFor('auth').get('isLoggedIn')) {
      this.transitionTo('login');
    }
  }
});
```

The `afterModel` hook

- Called after a route's model (which might be a promise) is resolved, and follows the same pause-on-promise semantics as `model` and `beforeModel`.
- It is passed the already-resolved model and can therefore perform any additional logic that depends on the fully resolved value of a model.

Example afterModel usage

```
App.ArticlesRoute = Ember.Route.extend({
  model: function() {
    // App.Article.find() returns a promise-like object
    // (it has a `then` method that can be used like a promise)
    return App.Article.find();
  },
  afterModel: function(articles) {
    if (articles.get('length') === 1) {
      this.transitionTo('article.show', articles.get('firstObject'));
    }
  }
});
```

Loading / Error States

Loading / Error States

Ember Router provides powerful yet overridable conventions for customizing asynchronous transitions between routes by making use of `error` and `loading` substates.

`loading` substates

- The Ember Router allows you to return promises from the various `beforeModel/model/afterModel` hooks in the course of a transition
- These promises pause the transition until they fulfill, at which point the transition will resume.

Consider the following:

```
App.Router.map(function() {
  this.resource('foo', function() { // -> FooRoute
    this.route('slowModel');        // -> FooSlowModelRoute
  });
});

App.FooSlowModelRoute = Ember.Route.extend({
  model: function() {
    return somePromiseThatTakesAWhileToResolve();
  }
});
```

Need Visual Feedback

- If you navigate to `foo/slow_model`, and in `FooSlowModelRoute#model`, you return an AJAX query promise that takes a long time to complete.
- During this time, your UI isn't really giving you any feedback as to what's happening
- Even worse if you are entering from a full page refresh as the UI will be completely blank
- If you navigate from another route, you'll see the old contents while loading

So, how can we provide some visual feedback during the transition?

Ember provides a default implementation of the `loading` process that implements the following loading substate behavior.

```
App.Router.map(function() {
  this.resource('foo', function() {       // -> FooRoute
    this.resource('foo.bar', function() { // -> FooBarRoute
      this.route('baz');                  // -> FooBarBazRoute
    });
  });
});
```

If a route with the path `foo.bar.baz` returns a promise that doesn't immediately resolve, Ember will try to find a `loading` route in the hierarchy above `foo.bar.baz` that it can transition into, starting with `foo.bar.baz`'s sibling:

1. `foo.bar.loading`
2. `foo.loading`
3. `loading`

Ember will find a loading route at the above location if either a) a Route subclass has been defined for such a route, e.g.

1. `App.FooBarLoadingRoute`
2. `App.FooLoadingRoute`
3. `App.LoadingRoute`

or b) a properly-named loading template has been found, e.g.

1. `foo/bar/loading`
2. `foo/loading`
3. `loading`

So, to fix slow asynchronous loading

- Ember will transition into the first loading sub-state/route that it finds, if one exists.
- The intermediate transition into the loading substate happens immediately (synchronously),
- The URL won't be updated
- Unlike other transitions that happen while another asynchronous transition is active, the currently active async transition won't be aborted.

So, to fix slow asynchronous loading

- After transitioning into a loading substate, the corresponding template for that substate, if present, will be rendered into the main outlet of the parent route
- e.g. `foo.bar.loading`'s template would render into `foo.bar`'s outlet. (This isn't particular to loading routes; all routes behave this way by default.)
- Once the main async transition into `foo.bar.baz` completes, the loading substate will be exited, its template torn down, `foo.bar.baz` will be entered, and its templates rendered.

Eager vs. Lazy Async Transitions

- Loading substates are optional
- If you provide one, you are essentially telling Ember that you want this async transition to be "eager"
- If you don't provide one, it will be "lazy" and remain on the pre-transition route until ready

This has implications on error handling, i.e. when a transition into another route fails, a lazy transition will (by default) just remain on the previous route, whereas an eager transition will have already left the pre-transition route to enter a loading substate.

The `loading` event

- If you return a promise from the various `beforeModel`/`model`/`afterModel` hooks, and it doesn't immediately resolve, a `loading` event will be fired on that route and bubble upward to `ApplicationRoute`.
- If the `loading` handler is not defined at the specific route, the event will continue to bubble above a transition's pivot route, providing the `ApplicationRoute` the opportunity to manage it.

The `loading` event

```
App.FooSlowModelRoute = Ember.Route.extend({
  model: function() {
    return somePromiseThatTakesAWhileToResolve();
  },
  actions: {
    loading: function(transition, originRoute) {
      //displayLoadingSpinner();

      // Return true to bubble this event to `FooRoute`
      // or `ApplicationRoute`.
      return true;
    }
  }
});
```

The `loading` event

The `loading` handler provides the ability to decide what to do during the loading process. If the last loading handler is not defined or returns `true`, Ember will perform the loading substate behavior.

```
App.ApplicationRoute = Ember.Route.extend({
  actions: {
    loading: function(transition, originRoute) {
      displayLoadingSpinner();

      // substate implementation when returning `true`
      return true;
    }
  }
});
```

`error` substates

- Ember provides an analogous approach to `loading` substates in the case of errors encountered during a transition.

```
App.Router.map(function() {
  this.resource('articles', function() { // -> ArticlesRoute
    this.route('overview');              // -> ArticlesOverviewRoute
  });
});
```

For instance, an error thrown or rejecting promise returned from `ArticlesOverviewRoute#model` (or `beforeModel` or `afterModel`) will look for:

1. Either `ArticlesErrorRoute` or `articles/error` template

2. Either `ErrorRoute` or `error` template

If one of the above is found, the router will immediately transition into that substate (without updating the URL). The "reason" for the error will be passed to that error state as its `model`.

If no viable error substates can be found, an error message will be logged.

error substates with dynamic segments

```
App.Router.map(function() {
  this.resource('foo', {path: '/foo/:id'}, function() {
    this.route('baz');
  });
});

App.FooRoute = Ember.Route.extend({
  model: function(params) {
    return new Ember.RSVP.Promise(function(resolve, reject) {
       reject("Error");
    });
  }
});
```

`error` substates with dynamic segments

- In the URL hierarchy you would visit `/foo/12` which would result in rendering the `foo` template into the `application` template's `outlet`.
- In the event of an error while attempting to load the `foo` route you would also render the top-level `error` template into the `application` template's `outlet`. - - This is intentionally parallel behavior as the `foo` route is never successfully entered.

In order to create a `foo` scope for errors and render `foo/error` into `foo`'s `outlet` you would need to split the dynamic segment:

```
App.Router.map(function() {
  this.resource('foo', {path: '/foo'}, function() {
    this.resource('elem', {path: ':id'}, function() {
      this.route('baz');
    });
  });
});
```

The `error` event

- If `ArticlesOverviewRoute#model` returns a promise that rejects (Perhaps from a server error)
- An `error` event will fire on `ArticlesOverviewRoute` and bubble upward.
- This `error` event can be handled and used to display an error message, redirect to a login page, etc.

```
App.ArticlesOverviewRoute = Ember.Route.extend({
  model: function(params) {
    return new Ember.RSVP.Promise(function(resolve, reject) {
       reject("Error");
    });
  },
  actions: {
    error: function(error, transition) {

      if (error && error.status === 400) {
        // error substate and parent routes do not handle this error
        return this.transitionTo('modelNotFound');
      }

      // Return true to bubble this event to any parent route.
      return true;
   }
  }
});
```

In like the `loading` event, you can manage the `error` event at the Application level:

```
App.ApplicationRoute = Ember.Route.extend({
  actions: {
    error: function(error, transition) {

      // Manage your errors
      Ember.onerror(error);

      // substate implementation when returning `true`
      return true;

    }
  }
});
```

Preventing and Retrying Transitions

Preventing and Retrying Transitions

- During a route transition, the Ember Router passes a transition object to the various hooks on the routes involved in the transition.
- Any hook that has access to this transition object has the ability to immediately abort the transition by calling `transition.abort()`, and if the transition object is stored, it can be re-attempted at a later time by calling `transition.retry()`.

Preventing Transitions via `willTransition`

- When a transition is attempted, whether via `{{link-to}}`, `transitionTo`, or a URL change, a `willTransition` action is fired on the currently active routes.
- This gives each active route, starting with the leaf-most route, the opportunity to decide whether or not the transition should occur.
- Imagine your app is in a route that's displaying a complex form for the user to fill out and the user accidentally navigates backwards. The user might lose data.

Here's one way this situation could be handled:

```
App.FormRoute = Ember.Route.extend({
  actions: {
    willTransition: function(transition) {
      if (this.controller.get('userHasEnteredData') &&
          !confirm("Are you sure you want to abandon progress?")) {
        transition.abort();
      } else {
        // Bubble the `willTransition` action so that
        // parent routes can decide whether or not to abort.
        return true;
      }
    }
  }
});
```

Aborting Transitions Within `model`, `beforeModel`, `afterModel`

- The `model`, `beforeModel`, and `afterModel` hooks get called with a transition object.
- This makes it possible for destination routes to abort attempted transitions.

```
App.DiscoRoute = Ember.Route.extend({
  beforeModel: function(transition) {
    if (new Date() < new Date("January 1, 1980")) {
      alert("Sorry, you need a time machine to enter this route.");
      transition.abort();
    }
  }
});
```

Storing and Retrying a Transition

- Aborted transitions can be retried at a later time.
- A common use case for this is having an authenticated route redirect the user to a login page, and then redirecting them back to the authenticated route once they've logged in.

```
App.SomeAuthenticatedRoute = Ember.Route.extend({
  beforeModel: function(transition) {
    if (!this.controllerFor('auth').get('userIsLoggedIn')) {
      var loginController = this.controllerFor('login');
      loginController.set('previousTransition', transition);
      this.transitionTo('login');
    }
  }
});

App.LoginController = Ember.Controller.extend({
  actions: {
    login: function() {
      // Log the user in, then reattempt previous transition if it exists.
      var previousTransition = this.get('previousTransition');
      if (previousTransition) {
        this.set('previousTransition', null);
        previousTransition.retry();
      } else {
        // Default back to homepage
        this.transitionToRoute('index');
      }
    }
  }
});
```

COEN 168/268, Winter 2014

Mobile Web Application Development

Ember Routing

Peter Bergström (pbergstrom@scu.edu)

Santa Clara University