ember-cli 101

by **Adolfo Builes**

ember-cli 101

Learn Ember.js with ember-cli.

Adolfo Builes

This book is for sale at http://leanpub.com/ember-cli-101

This version was published on 2015-03-30

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Adolfo Builes by spreading the word about this book on Twitter!

The suggested hashtag for this book is #ember-cli-101.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#ember-cli-101

Contents

Why

Before getting into the specifics, I'd like to explain why ember-cli was created and how it is different from other tools.

The main objective of ember-cli is to reduce what we call glue code and allow developers to focus on what is most important for them: building their app.

Glue code refers to those things that are not related to your application but that every project requires. For example, you need to test your code, compile your assets, serve your files in the browser, interact with a back-end API, perhaps use third party libraries, and so on. All those things can be automated and, as it is done in other frameworks, some conventions can provide a common ground to begin building your applications.

Having a tool that does that for you not only eases the process of writing your app but also saves you time and money (you don't have to think about problems that are already solved).

`ember-cli` aims to be exactly that tool. Thanks to `Broccoli`[1], waiting time is reduced while your assets compile. `QUnit`[2] allows you to write tests, which can then be run with `Testem`[3]. If you need to deploy your build to production, you'll get fingerprint, compression, and some other features for free.

`ember-cli` also encourages the use of ES6(ECMAScript 6)[4]. It provides built-in support for modules and integrates easily with other plugins, allowing you to write your applications using other ES6 features.

Next time you consider wasting your day wiring up all those things I just mentioned, consider `ember-cli`. It will make your life easier and you will get support from a lot of smart people who are already using this tool.

---

[1] https://github.com/broccolijs/broccoli

[2] http://qunitjs.com/

[3] https://github.com/airportyh/testem

[4] https://people.mozilla.org/~jorendorff/es6-draft.html

Anatomy

In this chapter we will learn about the main components of ember-cli.

ember-cli is a **Node.js** command line application that sits on top of other libraries.

Its main component is **Broccoli**, a builder designed to keep builds as fast as possible.

When we run `ember server`, **Broccoli** compiles our project and puts it in a directory where it can be served using **Express.js**[5], a **Node.js** library. **Express** not only serves files but also extends **ember-cli**'s functionality using its **middlewares**. An example of this is **http-proxy**, which supports the `--proxy` option that allows us to develop against our development backend.

Testing is powered by **QUnit** and **Testem**. By navigating to **http:/localhost:4200/tests**, our tests run automatically. We can also run Testem in **CI** or `--development` mode with the **ember test** command. Currently, only **QUnit** is supported and it's done via an **ember-cli add-on**. We will probably see support for other testing frameworks and runners as more people become familiar with the add-on system.

ember-cli uses its own resolver and has a different naming convention from **Ember.js's** defaults.

ember-cli makes us write our application using **ES6 Modules**. The code is then transpiled (compiled)[6] to **AMD**[7] and finally loaded with the minimalist **AMD**[8] loader, **loader.js**.

You can use **CoffeeScript** if you want, but you are encouraged to use plain JS and ES6 modules where possible. In subsequent chapters, we'll explore its syntax and features.

Finally, we need to cover plugins that enhance the functionality of **Broccoli**. Each transformation your files go through is done with a **Broccoli** plugin, e.g. transpiling, minifying, finger-printing, uglifying. You can have your own **Broccoli** plugins and plug them wherever you like throughout the build process.

---

[5] http://expressjs.com/

[6] The transpiling process is done with es6-module-transpiler.

[7] To know more about **AMD** checkout their wiki

[8] To know more about **AMD** checkout their wiki

Conventions

We will explore some of the basic conventions and best practices both in `Ember.js` and `ember-cli`.

In your code

- Use `camelCase` even if you are writing `CoffeeScript`.
- Avoid globals as much as possible: `ember-cli` supports `ES6 Modules` out of the box so you can write your app in a modular way.
- Create custom `shims` for apps that are not distributed in `AMD` format: we will cover this in a subsequent chapter, but the basic idea is to treat libraries that are not written with `ES6 Modules` as if they were.
- Create reusable code: if there is a functionality you are using in several different places, remember that `Ember.js` offers an Ember.Mixin class[9] that you can then reuse in different parts. If you think other people can benefit from this, create an add-on.

In your project

- Name your files using kebab-case: Use hyphens instead of underscores to separate words in a file name. For example, if you have a model called `InvoiceItem`, `ember-cli` expects this model to be under `app/models/invoice-item.js`.
- Optionally, include the file type at the beginning: Some people like to include the file type in the name of the file (e.g. `app/routes/route-index.js`). I personally prefer not to do this, but if you want to, just remember to include it at the beginning. Otherwise, your app will not be able to find (in this case) the `IndexRoute`.
- Put child files in subdirectories:
  - app/routes/invoice-item/index.js
  - app/controllers/invoice-items/index.js

[9] http://emberjs.com/api/classes/Ember.Mixin.html

Getting started

With this book, we'll create an app to keep track of items we lend to our friends. It's a very simple app, but it will allow us to learn Ember. At the same time, we'll learn how to use ember-cli generators, work with third party libraries, and write **ember-cli add-ons**.

Requirements

1. Install `Node.js`. The easiest way is to download the installer from http://nodejs.org/[10].
2. Install the ember-inspector. Click here for Chrome[11] or here for Firefox[12].
3. Install watchman[13] for fast watching. We can start it with `watchman watch ~/path-to-dir`.
4. Make sure you are not required to run `npm` (Node's package manager) with sudo. To test this, run the following command

```
npm -g install ember-cli
```

If you were prompted to install as sudo, make sure you can run `npm` without it. Tyler Wendlandt wrote an excellent tutorial for installing `npm` without sudo: http://www.wenincode.com/installing-node-jsnpm-without-sudo[14]. It's very important that you are not required to run `npm` as sudo, otherwise you will have problems when running ember-cli.

All set? Now let's create our first ember-cli app.

ember new

Like other command line tools, `ember-cli` comes with a bunch of useful commands. The first one we will explore is `new`, which creates a new project.

[10] http://nodejs.org/
[11] https://chrome.google.com/webstore/detail/ember-inspector/bmdblncegkenkacieihfhpjfppoconhi?hl=en
[12] https://addons.mozilla.org/en-US/firefox/addon/ember-inspector/
[13] https://github.com/facebook/watchman
[14] http://www.wenincode.com/installing-node-jsnpm-without-sudo

Creating a new project

```
ember new borrowers
```

The **new** command will create a directory with the following structure:

Project Structure

```
|-- Brocfile.js
|-- README.md
|-- app
|-- bower.json
|-- bower_components
|-- config
|-- node_modules
|-- package.json
|-- public
|-- testem.json
|-- tests
+-- vendor
```

We can add `--help` to any `ember` command to see available options (e.g., `ember new --help`).

By default, ember-cli assumes we are using git. If we are not, we can opt out by passing **–skip-git**: `ember new borrowers --skip-git`.

We will cover all the components as we move through this text, but the following are the most important.

- `app` is where the app code is located: controllers, routes, views, templates, and styles.
- `tests` is where test code is located.
- `bower.json` helps us manage `JavaScript` plugins via `Bower`.
- `package.json` helps us with `JavaScript` dependencies via `npm`.

> A question that pops up often is, "What's the difference between `npm` and `bower`?" From this[a] `Stack Overflow`: *Npm and Bower are both dependency management tools. The main difference between them is that npm is used to install Node js modules while bower js is used to manage front end components like html, css, js, etc.
>
> [a] http://stackoverflow.com/questions/21198977/difference-between-grunt-npm-and-bower-package-json-vs-bower-json/21199026#21199026

If everything is fine, we can do `ember server` and navigate to `http://localhost:4200` where we should see a `Welcome to Ember.js` message.

Hands-on

In the following sections we will add some models to our app, define the interactions between them, and create an interface to add friends and the articles they borrow from us.

Adding a friend resource

The main model of our application will be called **Friend**. It represents the people who will borrow articles from us.

Let's add it with the **resource** generator.

```
$ ember generate resource friends firstName:string lastName:string \
      email:string twitter:string totalArticles:number
  create app/models/friend.js
  create app/routes/friends.js
  create app/templates/friends.hbs
  create tests/unit/models/friend-test.js
  create tests/unit/routes/friends-test.js
```

If we open **app/models/friend.js** or **app/routes/friends.js**, we will see that they have a similar structure.

Object Structure

```
import  Foo from 'foo';

export default Foo.extend({
});
```

What is that? **ES6 Modules**! As mentioned previously, **ember-cli** expects us to write our code using ES6 Modules. `import Foo from 'foo'` consumes the default export from the package `foo` and assigns it to the variable `Foo`. We use `export default Foo.extend...` to define what our module will expose. In this case we will export a single value, which will be a subclass of `Foo`.

For a better understanding of ES6 modules, visit http://jsmodules.io/[15].

[15] http://jsmodules.io

Now let's look at the model and route.

app/models/friend.js

```
// We import the default value from ember-data into the variable DS.
//
// Ember-Data exports by default a namespace (known as DS) that exposes all the
// classes and functions defined in http://emberjs.com/api/data.

import DS from 'ember-data';

// Define the default export for this model, which will be a subclass
// of DS.Model.
//
// After this class has been defined, we can import this subclass doing:
// import Friend from 'borrowers/models/friend'
//
// We can also use relative imports. So if we were in another model, we
// could have written
// import Friend from './friend';

export default DS.Model.extend({

  // DS.attr is the standard way to define attributes with Ember-Data
  firstName: DS.attr('string'),


  // Defines an attribute called lastName of type **string**
  lastName: DS.attr('string'),


  // Ember-Data expects the attribute **email** on the friend's payload
  email: DS.attr('string'),

  twitter: DS.attr('string'),
  totalArticles: DS.attr('number')
});
```

app/routes/friends.js

```
// Assigns the default export from **ember** into the variable Ember.
//
// The default export for the ember package is a namespace that
// contains all the classes and functions for Ember that are specified in
// http://emberjs.com/api/

import Ember from 'ember';

// Defines the default export for this module. For now we will not
// add anything extra, but if we want to use a Route **hook** or
// **actions** this would be the place.

export default Ember.Route.extend({
});
```

In a future version of **Ember** we might be able to be more explicit about the things we want to use from every module. Instead of writing **import Ember from 'ember'**, we could have **import { Route } from 'ember'** or **import { Model } from 'ember-data'**. This is currently possible in **ES6** using Named Imports and Exports[16].

What about tests? If we open the test files, we'll see that they are also written in ES6. We'll talk about that in a later chapter. Now let's connect to a backend and display some data.

Connecting with a Backend

We need to consume and store our data from somewhere. In this case, we created a public API under **http://api.ember-cli-101.com** with **Ruby on Rails**. The following are the API end-points.

Verb	URI Pattern
GET	/api/articles
POST	/api/articles
GET	/api/articles/:id
PATCH	/api/articles/:id
PUT	/api/articles/:id
DELETE	/api/articles/:id
GET	/api/friends
POST	/api/friends
GET	/api/friends/:id
PATCH	/api/friends/:id

[16] http://jsmodules.io

Verb	URI Pattern
PUT	/api/friends/:id
DELETE	/api/friends/:id

If we do a **GET** request to **/api/friends**, we will get a list of all our friends.

```
# The following output might be different for every run since the data
# in the API is changing constantly.
#
$ curl http://api.ember-cli-101.com/api/friends.json | python -m json.tool
{
    "friends": [
        {
            "email": "test@gmail.com",
            "first_name": "jon",
            "id": 1,
            "last_name": "snow",
            "twitter": "foo"
        }
    ]
}
```

Piping JSON data to **python -m json.tool** is an easy way to pretty print JSON data in our console using python's JSON library. It's very useful if we want to quickly debug JSON data.

When returning a list, **Ember-Data** expects the root name of the JSON payload to match the name of the model but pluralized (**friends**) and followed by an array of objects. This payload will help us to populate **Ember-Data** store.

If we want to run the server by ourselves or create our own instance on **Heroku**, we can use the **Heroku Button** added to the repository borrowers-backend[17].

Once we have created our own instance on **Heroku**, we need to install Heroku Toolbelt[18] and check our application's log with `heroku logs -t --app my-app-name`.

A word on Adapters

By default, Ember-Data uses the **DS.RESTAdapter**[19], which expects everything to be in **camelCase** following **JavaScript**'s coding conventions. In our example, however, we will work with an API

[17] https://github.com/abuiles/borrowers-backend

[18] https://toolbelt.heroku.com/

[19] We recommend going through the documentation to get more insights on this adapter DS.RESTAdapter.

written in **Ruby on Rails** that uses a different convention for keys and naming. Everything is in **snake_case**.

We mentioned previously that everything has to be in **camelCase** since it is what the default **Ember-Data** adapter expects, but we can extend the **DS.RESTAdapter** to write our own adapter, matching our backend's payload.

This is such a common scenario that **Ember-Data** includes by default a **DS.ActiveModelAdapter**[20] that is modeled after **rails-api**'s project active_model_serializers[21]. This is widely used in the **Ruby on Rails** world and basically helps build the **JSON** that the API will return.

The following is the implementation of DS.ActiveModelAdapter[22]. It's just a few lines of code and it helps us understand what's going on under the hood.

There are a bunch of different adapters for different projects and frameworks. Some of them are:

- ember-data-django-rest-adapter[23]
- ember-data-tastypie-adapter[24]
- emberfire: FireBase adapter[25]

We can find a longer list of adapters if we search GitHub for ember-data adapters[26].

Specifying our own adapter

As mentioned in the previous chapter, if we are using **Ember-Data** it will **resolve** to the **DS.RESTAdapter** unless we specify something else.

To see it in action, let's play with the console and examine how **Ember** tries to **resolve** things.

First we need to go to `config/environment.js` and uncomment `ENV.APP.LOG_RESOLVER`[27]. It should look like:

[20] Documentation for DS.ActiveModelAdapter.html.
[21] https://github.com/rails-api/active_model_serializers
[22] https://github.com/emberjs/data/blob/v1.0.0-beta.10/packages/activemodel-adapter/lib/system/active_model_adapter.js#L104
[23] https://github.com/toranb/ember-data-django-rest-adapter
[24] https://github.com/escalant3/ember-data-tastypie-adapter
[25] https://github.com/firebase/emberfire
[26] https://github.com/search?q=ember-data+adapter&ref=opensearch
[27] Enable ENV.APP.LOG_RESOLVER.

config/environment.js

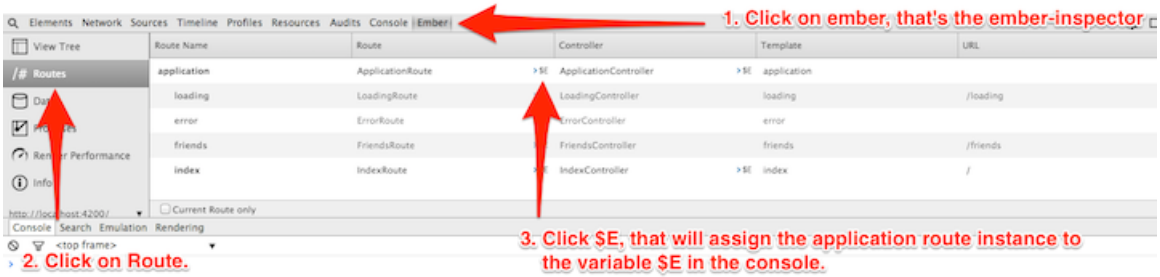
```
if (environment === 'development') {
  ENV.APP.LOG_RESOLVER = true;
  ENV.APP.LOG_ACTIVE_GENERATION = true;
  // ENV.APP.LOG_TRANSITIONS = true;
  // ENV.APP.LOG_TRANSITIONS_INTERNAL = true;
  ENV.APP.LOG_VIEW_LOOKUPS = true;
}
```

That line will log whatever **Ember** tries to "find" to the browser's console. If we stop `ember server`, start it again, go to http://localhost:4200[28], click refresh, and open the console, we'll see:

```
[ ] router:main ............... borrowers/main/router
[ ] router:main ............... borrowers/router
[✓] router:main ............... borrowers/router
[ ] application:main ......... borrowers/main/application
[ ] application:main ......... undefined
[ ] application:main ......... borrowers/application
[ ] application:main ......... borrowers/applications/main
[ ] application:main ......... undefined
```

That's the **Ember** resolver trying to find things. We don't need to worry about understanding all of it right now.

Coming back to the **Adapter**, if we open the **ember-inspector** and grab the instance of the **Application** route



ember-inspector

[28] http://localhost:4200

We can grab almost any instance of a Route, Controller, View or Model with the **ember-inspector** and then reference it in the console with the `$E` variable. This variable is reset every time the browser gets refreshed.

With the **ApplicationRoute** instance at hand, let's have some fun.

Let's examine what happens if we try to find all our **friends**:

```
$E.store.findAll('friend')
[ ] adapter:friend ............borrowers/friend/adapter
[ ] adapter:friend ............undefined
[ ] adapter:friend ............borrowers/adapters/friend
[ ] adapter:friend ............undefined
[ ] adapter:application ........borrowers/application/adapter
[ ] adapter:application ........undefined
[ ] adapter:application ........borrowers/adapters/application
[ ] adapter:application ........undefined
```

First, the **Resolver** tries to find an adapter at the model level:

```
[ ] adapter:friend ............borrowers/friend/adapter
[ ] adapter:friend ............undefined
[ ] adapter:friend ............borrowers/adapters/friend
[ ] adapter:friend ............undefined
```

We can use this if we want to change the default behavior of **Ember-Data**. For example, changing the way an URL is generated for a resource.

Suppose a friend `hasMany('article')` and we are using nested URLs in the backend. In this case, the URL for an article will be **/friends/1/articles/1** instead of **articles/1**

We can fix this overriding buildURL[29]:

[29] http://emberjs.com/api/data/classes/DS.RESTAdapter.html#method_buildURL

Custom adapter for a model called article: app/adapters/article.js

```
export default ApplicationAdapter.extend({
  // This is only an example of how buildURL can be used, but we
  // actually don't use this in our app, please don't create this
  // adapter.

  buildURL: function(type, id, record) {
    return '/friends/' + record.get('friend.id') + '/articles/' + id;
  }
})
```

Second, if no adapter is specified for the model, then the **Resolver** checks if we specified an **Application** adapter. As we can see, it returns **undefined**, which means we didn't specify one:

```
[ ] adapter:application ........borrowers/application/adapter
[ ] adapter:application ........undefined
[ ] adapter:application ........borrowers/adapters/application
[ ] adapter:application ........undefined
```

Third, if no model or application adapter is found, then **Ember-Data** falls back to the default adapter, the **RESTAdapter**. We can check the implementation for this directly in the adapterFor[30] function in **Ember-Data**.

We can see that there is a look up for the friend and application adapter in two places **borrowers/friend/adapter**, **borrowers/adapters/friend**, **borrowers/application/adapter** and **borrowers/adapters/application**. ember-cli allows us to group things that are logically related under a single directory. This structure is known as PODS. We'll work with the normal structure first, and at the end of the book we'll rewrite a part of our code to be structured under PODS.

Since we want to work with a different adapter, we need to tell **Ember** to do so. In this case we want the **DS.ActiveModelAdapter** as our application adapter. Again, **ember-cli** has a generator for adapters.

ember g is a short version of **ember generator**. We'll use both interchangeably to get used to the syntax.

Run `ember g adapter application` to create an application adapter:

[30] https://github.com/emberjs/data/blob/131119/packages/ember-data/lib/system/store.js#L1552

```
$ ember g adapter application
version: 0.1.9
installing
  create app/adapters/application.js
installing
  create tests/unit/adapters/application-test.js
```

It will create a file like the following:

app/adapters/application.js

```
import DS from 'ember-data';

export default DS.RESTAdapter.extend({
});
```

But we don't want to use the **DS.RESTAdapter** so let's change that file to look like the following:

app/adapters/application.js

```
import DS from 'ember-data';

export default DS.ActiveModelAdapter.extend({
  namespace: 'api'
});
```

We now specify our **Adapter** and also pass a property **namespace**. The **namespace** option tells **Ember-Data** to namespace all our **API** requests under `api`. So if we ask for the collection `friend`, **Ember-Data** will make a request to `/api/friends`. If we don't have that, then it will be just `/friends`.

Let's go back to our browser's console, grab the **ApplicationRoute** instance again from the **ember-inspector**, and ask the store for our friends.

```
$E.store.findAll('friend')
[ ] adapter:friend ............ borrowers/friend/adapter
[ ] adapter:friend ............ undefined
[ ] adapter:friend ............ borrowers/adapters/friend
[ ] adapter:friend ............ undefined
[ ] adapter:application ........ borrowers/application/adapter
[ ] adapter:application ........ borrowers/adapters/application
[✓] adapter:application ........ borrowers/adapters/application
[✓] adapter:application ........ borrowers/adapters/application
[✓] adapter:application ........ borrowers/adapters/application
GET http://localhost:4200/api/friends 404 (Not Found)
```

This time, when the **Resolver** tries to find an adapter, it works because we have one specified under **applications/adapters**. We also see a failed **GET** request to **api/friends**. It fails because we are not connected to any backend yet.

Stop the **ember server** and start again, but this time let's specify that we want all our **API** requests to be proxy to **http://api.ember-cli-101.com**. To do so we use the option **–proxy**:

Running ember server

```
$ ember server --proxy http://api.ember-cli-101.com
version: 0.1.9
Proxying to http://api.ember-cli-101.com
Livereload server on port 35729
Serving on http://0.0.0.0:4200
```

Go back to the console and load all our friends, but this time let's log something with the response:

```
$E.store.findAll('friend').then(function(friends) {
  friends.forEach(function(friend) {
    console.log('Hi from ' + friend.get('firstName'));
  });
});

XHR finished loading: GET "http://localhost:4200/api/friends".
Hi from jon
```

If we see 'Hi from' followed by a name, we have successfully specified our application adapter and connected to the backend. The output might be different every time we run it since the API's data is changing.

We use the name of our model in singular form. This is important. We always reference the models in their singular form.

Listing our friends

Now that we have successfully specified our own **Adapter** and made a request to our **API**, let's display our friends.

By convention, the entering point for rendering a list of any kind of resource in web applications is called the **Index**. This normally matches to the **Root** URL of our resource. With our friends example, we do so on the backend through the following end-point http://api.ember-cli-101.com/api/friends.json[31]. If we visit that URL, we will see a **JSON** list with all our friends.

If we are using Firefox or Chrome, we can use JSONView to have a readable version of **JSON** in our browser. Firefox Version[32] or Chrome Version[33].

In our Ember application, we need to specify somehow that every time we go to URL **/friends**, then all our users should be loaded and displayed in the browser. To do this we need to specify a **Route**.

Routes[34] are one of the main parts of **Ember**. They are in charge of everything related to setting up state, bootstrapping objects, specifying which template to render, etc. In our case, we need a **Route** that will load all our friends from the **API** and then make them available to be rendered in the browser.

Creating our first Route.

First, if we go to **app/router.js**, we will notice that the **resource** generator added **this.resource('friends', function() { });**.

[31] http://api.ember-cli-101.com/api/friends
[32] http://jsonview.com
[33] https://chrome.google.com/webstore/detail/jsonview/chklaanhfefbnpoihckbnefhakgolnmc
[34] http://emberjs.com/api/classes/Ember.Route.html

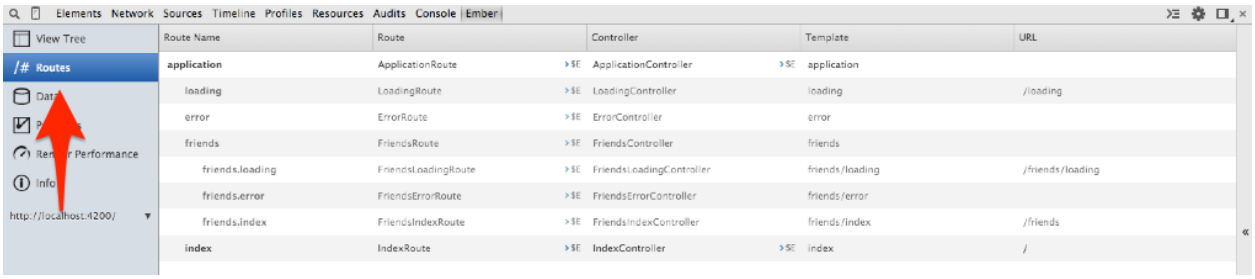
app/router.js

```
// ...

Router.map(function() {
  this.resource('friends', function() { });
});

// ...
```

We specify the **URLs** we want in our application inside the function passed to **Router.map**. There, we can call **this.route** or **this.resource**. The rule is: if we want a simple page that is not necessarily related with a resource, we use **this.route**. Otherwise, we use **this.resource**.

To know more about what a resource is, we recommend the following article on resources[35].

Let's check the **Routes** that we have currently defined. To do so, open the **ember-inspector** and click on **Routes**.



ember-inspector

By default, **Ember** creates 4 routes:

- ApplicationRoute
- IndexRoute
- LoadingRoute
- ErrorRoute

We also see that the **FriendsRoute** and its children were added with **this.resource('friends', function() { })**. **Ember** will create an **Index**, **Loading**, and **Error Route** if we pass a function as second or third argument.

[35] http://restful-api-design.readthedocs.org/en/latest/resources.html#resources

If we have defined the resource as **this.resource('friends')**, leaving out the empty function, then the children won't have been generated.

Since we have a **FriendsIndexRoute**, visiting http://localhost:4200/friends[36] should be enough to list all our friends. But if we actually go there, the only thing we will see is a message with **Welcome to Ember**.

Let's go to **app/templates/friends.hbs** and change it to look like the following:

app/templates/friends.hbs

```
<h1>Friends Route</h1>
{{outlet}}
```

For people familiar with Ruby on Rails, **{{outlet}}** is very similar to the word **yield** in templates. Basically it allows us to put content into it. If we check the application templates (**app/templates/application.hbs**), we'll find the following:

app/templates/application.hbs

```
<h2 id='title'>Welcome to Ember</h2>

{{outlet}}
```

When Ember starts, it will render the **Application Template** as the main template. Inside **{{outlet}}**, it will render the template associated with the **Route** we are visiting. Then, inside those templates, we can have more **{{outlet}}** to keep rendering content.

In our friends scenario, **app/templates/friends.hbs** will get rendered into the application's template **{{outlet}}**, and then it will render the **Friends Index** template into **app/templates/friends.hbs {{outlet}}**.

To connect everything, let's create an index template and list all our friends. Let's run the route generator **ember g route friends/index** and put the following content inside **app/templates/friends/index.hbs**:

[36] http://localhost:4200/friends

app/templates/friends/index.hbs

```
<h1>Friends Index</h1>

<ul>
  {{#each friend in model}}
     <li>{{friend.firstName}} {{friend.lastName}}</li>
  {{/each}}
</ul>
```

We remove **{{outlet}}** from **app/templates/friends/index.hbs** since the **Friends Index Route** won't have any nested route.

Next, we need to specify in the **Friends Index Route** the data we want to load in this route. The part in charge of loading the data related to a route is called the model hook. Let's add one to **app/routes/friends/index.js** as follows:

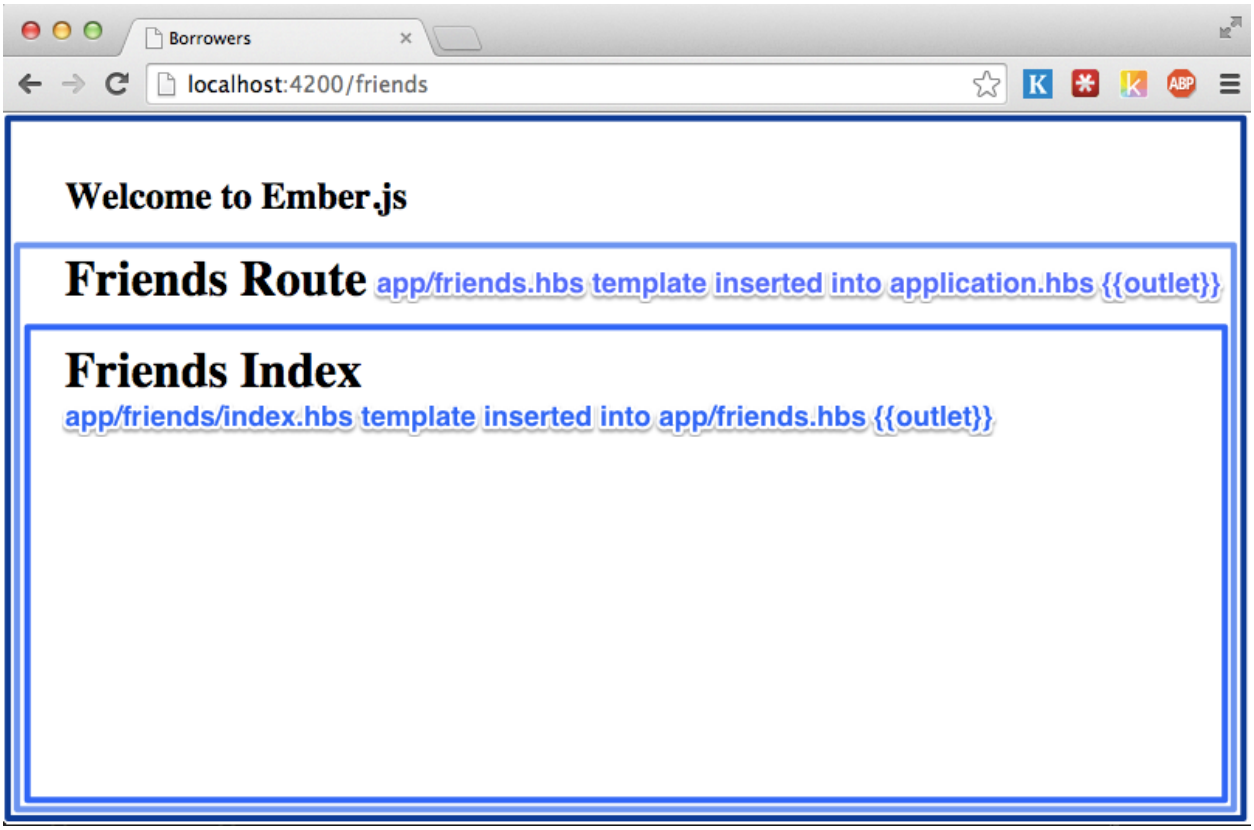
app/routes/friends/index.js

```
import Ember from 'ember';

export default Ember.Route.extend({
  model: function() {
    return this.store.findAll('friend');
  }
});
```

Remember that the **Route** is responsible for everything related to setting up the application state.

If we visit http://localhost:4200/friends[37] we will see something like the following along with a list of our friends:

[37] http://localhost:4200/friends



outlets

We played previously with **store.findAll** to load all our friends from the **API** and that's what we are doing in the model hook. **Ember** waits for this call to be completed. When the data is loaded, it automatically creates a **Friends Index Controller** (or we can define a controller explicitly) and sets the property **model** with the content returned from the **API**.

We can also use `store.find` or `store.findQuery` if we want to load a record by a given id or appending query parameters to the request URL, such as **this.store.find('friend', 1)** or **this.store.findQuery('friend', {active: true})**, ending in the following requests to the API **/api/friends/1** or **/api/friends?active=true**.

When we do **{{#each friend in model}}**, Ember (under the hood) takes every element of the collection and set it as **friend**, the collection which is what the model hook returned is referenced as **model**.

If we want to display the total number of friends and the **id** for every friend, then we just need to reference **model.length** in the template and inside the each use **friend.id**:

app/templates/friends/index.hbs

```
<h1>Friends Index</h1>
{{! The context here is the controller}}
<h2>Total friends: {{model.length}}</h2>

<ul>
  {{#each friend in model}}
    <li>{{friend.id}} - {{friend.firstName}} {{friend.lastName}}</li>
  {{/each}}
</ul>
```

Again, because our model is a collection and it has the property **length**, we can just reference it in the template as **model.length**.

Adding a new friend

We are now able to see which friends have borrowed things from us, but we don't have a way to add new friends. The next step is to build support for adding a new friend.

To do this we'll need a **Friends New Route** under the resource friends, which will handle the URL **http://localhost:4200/friends/new**.

By convention, the URL for adding a new resource is **/resource_name/new**. For editing a resource, use **/resource_name/:resource_id/edit** and for showing a resource, use **/resource/:resource_id**.

To add the new route, run the **Route** generator with the parameters **friends/new**:

```
$ ember g route friends/new
installing
  create app/routes/friends/new.js
  create app/templates/friends/new.hbs
  create tests/unit/routes/friends/new-test.js
```

If we go to **app/router.js** we'll see that the **new** route was nested under the resource **friends**:

app/router.js

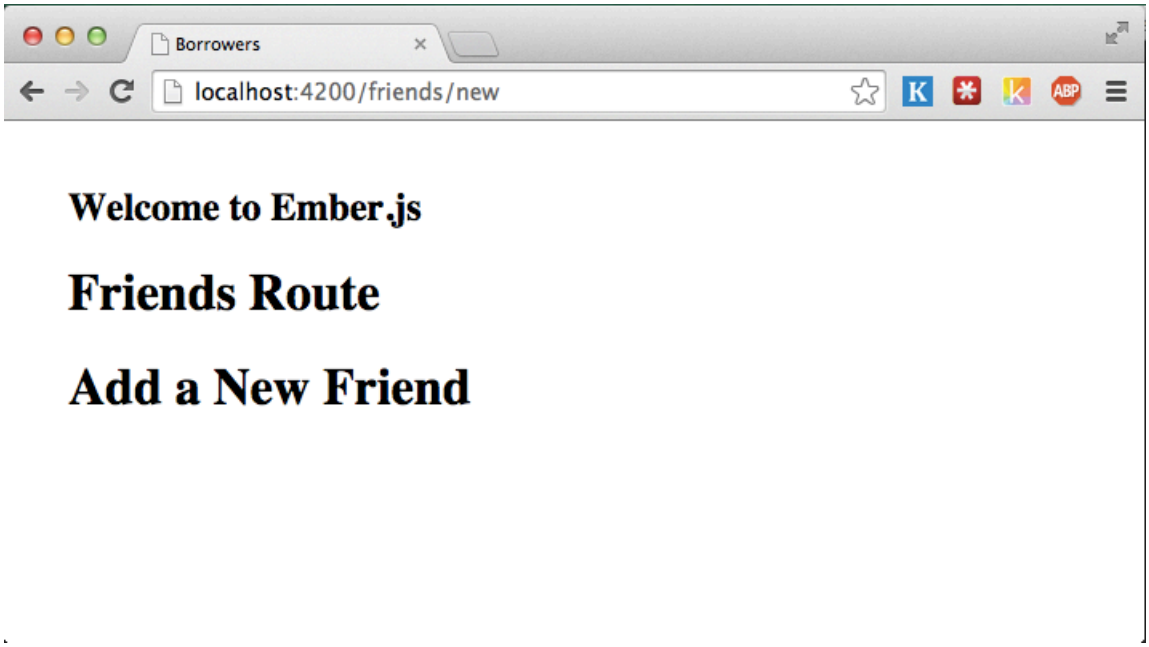
```
this.resource('friends', function(){
  this.route('new');
});
```

Add the following content on the new template:

app/templates/friends/new.hbs

```
<h1>Add a New Friend</h1>
```

And then navigate to http://localhost:4200/friends/new:



FriendsNewRoute

Notice how the **Friends New Route** got rendered in the **{{outlet}}** inside **app/templates/friends.hbs**.

We got our **Route** and **Template** wired up, but we can't add friends yet. We need to set a new friend instance as the model of the **Friends New Route**, create a form that will bind to the friend's attributes, and save the new friend in our backend.

Following the logic we used in the **Friends Index Route**, we need to return the model that will be the context of the **Friends New Route**. On the **model** hook function, go to **app/routes/friends/new.js** and add the following model hook:

```
import Ember from 'ember';

export default Ember.Route.extend({
  model: function() {
    return this.store.createRecord('friend');
  }
});
```

We have been using the **this.store** without knowing what it is. The Store[38] is an **Ember-Data** class in charge of managing everything related to our model's data. It knows about all the records we currently have loaded in our application and it has some functions that will help us to find, create, update, and delete records. During the whole application life cycle there is a unique instance of the **Store**, and it is injected as a property into every **Route**, **Controller**, **Serializer**, and **Adapter** under the key **store**. That's why we have been calling **.store** in our **Routes** and **Controllers**.

The following shows how the store is injected in every instance: store_injections[39].

The method we are using on the model hook **store.createRecord** creates a new record in our application **store**, but it doesn't save it to the backend. What we will do with this record is set it as the **model** of our **Friends New Route**. Then, once we have filled the first and last names, we can save it to our backend calling the method #save() in the model.

Since we will be using the same form for adding a new friend and editing, let's create an Ember partial[40] we can generate the template for the partial with template generator, `ember g template friends/-form` and add the following content:

app/templates/friends/-form.hbs

```
<form {{action "save" on="submit"}}>
  <p>
    <label>First Name:
      {{input value=model.firstName}}
    </label>
  </p>
  <p>
    <label>Last Name:
      {{input value=model.lastName }}
    </label>
  </p>
```

[38] http://emberjs.com/api/data/classes/DS.Store.html
[39] https://github.com/emberjs/data/blob/v1.0.0-beta.10/packages/ember-data/lib/initializers/store_injections.js
[40] http://emberjs.com/api/classes/Ember.Handlebars.helpers.html#method_partial

```
  <p>
    <label>Email:
      {{input value=model.email}}
    </label>
  </p>
  <p>
    <label>Twitter
      {{input value=model.twitter}}
    </label>
  </p>
  <input type="submit" value="Save"/>
  <button {{action "cancel"}}>Cancel</button>
</form>
```

As we mentioned in conventions, we should always use kebab-case when naming our files. This applies the same way to partials. In ember-cli, they should start with a dash followed by the partial name (**-form.hbs**). This is different from what Ember's website suggests, which is using an underscore.

Then we should modify the template **app/templates/friends/new.hbs** to include the partial:

app/templates/friends/new.hbs

```
<h1>Adding New Friend</h1>
{{partial "friends/form"}}
```

Now if we visit **http://localhost:4200/friends/new**, the form should be displayed.

There are some new concepts in what we just did. Let's talk about them.

Partials

In **app/templates/friends/new.hbs** we used

Using partials in app/templates/friends/new.hbs

```
{{partial "friends/form"}}
```

The **partial** method is part of the Ember.Handlebars.helpers[41] class. It is used to render other templates in the context of the current template. In our example, the friend form is a perfect candidate for a partial since we will be using the same form to create and edit a new friend.

[41] http://emberjs.com/api/classes/Ember.Handlebars.helpers.html#method_partial

{{action}}

The **{{action}}** helper is one of the most useful features in Ember. It allows us to bind an action in the template to an action in the template's **Controller** or **Route**. By default it is bound to the click action, but it can be bound to other actions.

The following button will call the action **cancel** when we click it.

```
<button {{action "cancel"}}>Cancel</button>
```

And **<form {{action "save" on="submit"}}>** will call the action **save** when the **onsubmit** event is fired; that is, when we click **Save**.

We could have written the save action as part of the submit button, but for demonstration purposes we put it in the form's **on="submit"** event.

If we go to the browser **http://localhost:4200/friends/new**, open the console, and click **Save** and **Cancel**, we'll see two errors. The first says **Nothing handled the action 'save'** and the second **Nothing handled the action 'cancel'**.

Ember expects us to define our action handlers inside the property **actions** in the **Controller** or **Route**. When the action is called, Ember first looks for a definition in the **Controller**. If there is none, it goes to the **Route** and keeps bubbling until **Application Route**. If any of the actions returns **false**, then it stops bubbling.

Let's create a controller for the **Friends New Route** and add the actions **save** and **cancel**.

To generate the **Friends New Controller**, we'll run `ember g controller friends/new` and then edit **app/controllers/friends/new.js** to add the property **actions**.

app/controllers/friends/new.js

```
import Ember from 'ember';

export default Ember.Controller.extend({
  actions: {
    save: function() {
      console.log('+- save action in friends new controller');

      return true;
    },
    cancel: function() {
      console.log('+- cancel action in friends new controller');
```

```
      return true;

    }
  }
});
```

If we go to **http://localhost:4200/friends/new** and click save, we'll see in the browser's console "**save action controller**".

Let's check next how returning **true** from the action makes it bubble. Go to **app/routes/friends/new.js** and add:

app/routes/friends/new.js

```
  actions: {
    save: function() {
      console.log('+-- save action bubbled up to friends new route');

      return true;
    },
    cancel: function() {
      console.log('+-- cancel action bubbled up to friends new route');

      return true;
    }
  }
```

Add in **app/routes/friends.js**:

app/routes/friends.js

```
actions: {
    save: function() {
      console.log('+--- save action bubbled up to friends route');

      return true;
    },
    cancel: function() {
      console.log('+--- cancel action bubbled up to friends route');

      return true;
    }
  }
```

And then create the file **app/routes/application.js** with:

app/routes/application.js

```
import Ember from 'ember';

export default Ember.Route.extend({
  actions: {
    save: function() {
      console.log('+---- save action bubbled up to application route');

      return true;
    },
    cancel: function() {
      console.log('+---- cancel action bubbled up to application route');

      return true;
    }
  }
});
```

After adding actions in all those routes, if we click **save** or **cancel** we'll see the action bubbling through every route currently active.

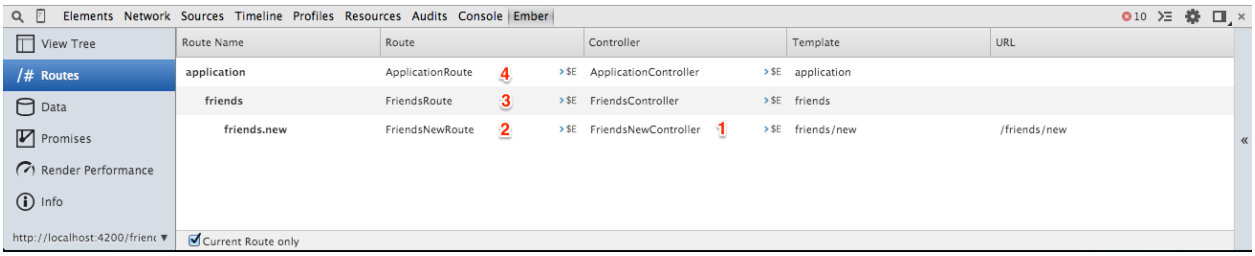
```
+- save action in friends new controller
+-- save action bubbled up to friends new route
+--- save action bubbled up to friends route
+---- save action bubbled up to application route
```

Again, it is bubbling because we are returning true from every child **actions**. If we want the action to stop bubbling, let's say in the **Friends Route**, we just need to return **false** in the actions specified in **app/routes/friends.js** and we'll get:

```
+- save action in friends new controller
+-- save action bubbled up to friends new route
+--- save action bubbled up to friends route
```

As we can see, the action didn't bubble up to the **Application Route**.

Whenever we have trouble understanding how our actions are going to bubble, we can go to the **ember-inspector**, click Routes, and then select **Current Route only**:



Actions Bubbling

As we can see, the action will bubble in the following order:

```
1. FriendsNewController
2. FriendsNewRoute
3. FriendsRoute
4. ApplicationRoute
```

How is this related to creating a new friend in our API? We'll discover that after we cover the next helper. Basically, on the **save** action, we'll validate our model, call **.save()**, which saves it to the API, and finally transition to a route where we can add new articles.

The input helper

Last we have the input helper[42]. It allows us to automatically bind a html input field to a property in our model. With the following **{{input value=firstName}}**, changing the value changes the property **firstName**.

If we add the following before the input buttons in **app/templates/friends/-form.hbs**

app/templates/friends/-form.hbs

```
<div>
  <h2>Friend details</h2>
  <p>{{model.firstName}}</p>
  <p>{{model.lastName}}</p>
</div>
```

And then go to the browser, we'll see that every time we change the first or last name field, this will change the description in **Friend details**.

We can also use the input helper to render other types of input such as a checkbox[43]. To do so, simply specify **type='checkbox'**.

[42] http://emberjs.com/api/classes/Ember.Handlebars.helpers.html#method_input
[43] http://emberjs.com/api/classes/Ember.Handlebars.helpers.html#toc_use-as-checkbox

```
{{input type="checkbox" name=trusted}}
```

If we click the checkbox, the attribute trusted will be true. Otherwise, it will be false.

Save it!

We learned about actions, **{{partial}}**, and **{{input}}**. Now let's save our friend to the backend.

To do so, we are going to validate the presence of all the required fields. If they are present, call **.save()** on the model. Otherwise, we'll see an error message on the form.

First we'll modify **app/templates/friends/-form.hbs** to include a field **{{errorMessage}}**.

app/templates/friends/-form.hbs

```
<form {{action "save" on="submit"}}>
  <h2>{{errorMessage}}</h2>
```

We will see the error every time we try to save a record without first filling in all the fields.

Then we'll implement a naive validation in **app/controllers/friends/new.js** by adding a computed property called **isValid**:

app/controllers/friends/new.js

```
export default Ember.Controller.extend({
  isValid: Ember.computed(
    'model.email',
    'model.firstName',
    'model.lastName',
    'model.twitter',
    function() {
      return !Ember.isEmpty(this.get('model.email')) &&
        !Ember.isEmpty(this.get('model.firstName')) &&
        !Ember.isEmpty(this.get('model.lastName')) &&
        !Ember.isEmpty(this.get('model.twitter'));
    }
  ),
  actions: {
   ....
  }
});
```

Ember.computed? That's new! Ember allows us to create functions that will be treated as properties. These are called computed properties. In our example, **isValid** is a **computed property** that depends on the properties **model.email**, **model.firstName**, **model.lastName**, and **model.twitter**. When any of those properties changes, the function that we passed-in is called and the value of our property is updated with the returned value.

In our example, we are manually checking that all the fields are not empty by using the isEmpty[44] helper.

With our naive validation in place, we can now modify our save and cancel actions:

actions in app/controllers/friends/new.js

```
save: function() {
  if (this.get('isValid')) {
    var _this = this;
    this.get('model').save().then(function(friend) {
      _this.transitionToRoute('friends.show', friend);
    });
  } else {
    this.set('errorMessage', 'You have to fill all the fields');
  }

  return false;
},
cancel: function() {
  this.transitionToRoute('friends');

  return false;
}
```

We might wonder why we are creating a copy of **this** in the variable **_this**. The reason is that we need to make a copy of **this** since the scope inside the function passed to **then** will be different. For more info in JavaScript's scope, read the blog post Scope and this in JavaScript[45].

When the action **save** is called, we are first checking if **isValid** is true. If it is, then we get the model and call **.save()**. The return of **save()** is a promise, which allow us to write asynchronous code in a sync manner. The function **.then** receives a function that will be called when the model has been saved successfully to the server. When this happens, it returns an instance of our friend and then we can transition to the route **FriendsShowRoute** to see our friend's profile.

[44] http://emberjs.com/api/classes/Ember.html#method_isEmpty

[45] http://javascriptplayground.com/blog/2012/04/javascript-variable-scope-this/

If we click save and have filled all the required fields, we'll still get an error: `The route friends.show was not found`. This is because we haven't defined a **Friends Show Route**. We'll do that in the next chapter.

For a better understanding of promises, I recommend the following talks from Ember NYC called The Promise Land[46].

Whenever we want to access a property of an Ember Object, we need to use **this.get('propertyName')**. It's almost the same as doing **object.propertyName**, but it adds extra features like handling computed properties. If we want to change the property of an object, we use **this.set('propertyName', 'newvalue')**. Again, it's almost equivalent to doing **this.propertyName = 'newValue'**, but it adds support so the observers and computed properties that depend on the property are updated accordingly.

Viewing a friend profile

Let's start by creating a **Friends Show Route**

```
$ ember g route friends/show --path=:friend_id
version: 0.1.9
installing
  create app/routes/friends/show.js
  create app/templates/friends/show.hbs
  create tests/unit/routes/friends/show-test.js
```

Route Generator

When creating a new route or resource we can use the route generator which takes the options `--type` and `--path`. With `type` we can use `route` or `resource`, with route being the default. We can see the options for every generator with `ember generate route --help`

If we open **app/router.js**, we'll see the route **show** nested under **friends**.

[46] https://www.youtube.com/watch?v=mZHO1ZTsoFk#t=2439

app/router.js

```
this.resource('friends', function(){
  this.route('new');
  this.route('show', { path: ':friend_id' });
});
```

We have talked previously about **path** but not about dynamic segments. **path: ':friend_id'** is specifying a dynamic segment. This means that our route will start with **/friends/** followed by an id that will be something like **/friends/12** or **/friends/ned-stark**. Whatever we pass to the URL, it will be available on the model hook under **params**, so we can reference it like **params.friend_id**. This will help us to load a specific friend by visiting the URL **/friends/:friend_id**. A route can have any number of dynamic segments (e.g., **path: '/friends/:group_id/:friend_id'**.)

Now that we have a **Friends Show Route**, let's start first by editing the template in **app/templates/friends/show.hbs**:

app/templates/friends/show.hbs

```
<ul>
  <li>First Name: {{model.firstName}}</li>
  <li>Last Name: {{model.lastName}}</li>
  <li>Email: {{model.email}}</li>
  <li>twitter: {{model.twitter}}</li>
</ul>
```

According to what we have covered, the next logical step would be to add a model hook on the **Friends Show Route** by calling **this.store.find('friend', params.friend_id)**. However, if we go to http://localhost:4200/friends/new and add a new friend, we'll be redirected to the **Friends Show Route** and our friend will be loaded without requiring us to write a model hook.

Why? As we have said previously, Ember is based on convention over configuration. The pattern of having dynamic segments like **model_name_id** is so common that if the dynamic segment ends with **_id**, then the model hook is generated automatically and it calls **this.store('model_name', params.model_name_id)**.

Visiting a friend profile

We can navigate to http://localhost:4200/friends to see all of our friends, but we don't have a way to navigate to their profiles!

Fear not. Ember has a helper for that as well, and it is called **{{link-to}}**.

Let's rewrite the content on **app/templates/friends/index.hbs** to use the helper:

app/templates/friends/index.hbs

```
{{#each friend in model}}
  <li>
    {{#link-to 'friends.show' friend}}
      {{friend.firstName}} {{friend.lastName}}
    {{/link-to}}
  </li>
{{/each}}
```

When we pass our intended route and an instance of a friend to **link-to**, it maps the property **id** to the parameter **friend_id**(we could also pass **friend.id**). Then, inside the block, we render the content of our link tag, which would be the first and last name of our friend.

One important item to mention is that if we pass an instance of a friend to **link-to**, then the model hook in the **Friends Show Route** won't be called. If we want the hook to be called, instead of doing `{{#link-to 'friends.show' friend}}`, we'll have to do `{{#link-to 'friends.show' friend.id}}`.

Check this example in JS BIN http://emberjs.jsbin.com/bupay/2/ that shows the behavior of **link-to** with an object and with an id.

The resulting HTML will look like the following

Output for link-to helper

```
<a id="ember476" class="ember-view" href="/friends/1">
 Jon Snow
</a>
```

If our friend model had a property called **fullName**, we could have written the helper like:

Using a computed for the link content

```
  {{link-to friend.fullName 'friends.show' friend}}
```

We already talked about computed properties, so let's add one called **fullName** to **app/models/friend.js**

app/models/friend.js

```
import DS from 'ember-data';
import Ember from 'ember';

export default DS.Model.extend({
  firstName: DS.attr('string'),
  lastName: DS.attr('string'),
  email: DS.attr('string'),
  twitter: DS.attr('string'),
  totalArticles: DS.attr('number'),
  fullName: Ember.computed('firstName', 'lastName', function() {
    return this.get('firstName') + ' ' + this.get('lastName');
  })
});
```

The computed property depends on **firstName** and **lastName**. Any time either of those properties changes, so will the value of **fullName**.

Once we have the computed property, we can rewrite **link-to** as follows:

Using friend.fullName in app/templates/friends/index.hbs

```
{{link-to friend.fullName 'friends.show' friend}}
```

Now we'll be able to visit any of our friends! Next, let's add support to edit a friend.

Quick Task

1. Add a link so we can move back and forth between a friend's profile and the friends index.
2. Add a link so we can move from **app/templates/index.hbs** to the list of friends (might need to generate the missing template).

Updating a friend profile

By now it should be clear what we need to update a friend:

1. Create a route with the **ember generator**.
2. Fix path in routes.
3. Update the template.
4. Add Controller and actions.

To create the **Friends Edit Route** we should run:

```
$ ember g route friends/edit --path=:friend_id/edit
version: 0.1.9
installing
  create app/routes/friends/edit.js
  create app/templates/friends/edit.hbs
installing
  create tests/unit/routes/friends/edit-test.js
```

The nested route **edit** should looks as follows under the the resource **friends**:

app/router.js

```
  this.resource('friends', function(){
    this.route('new');
    this.route('show', { path: ':friend_id' });
    this.route('edit', { path: ':friend_id/edit' });
  });
```

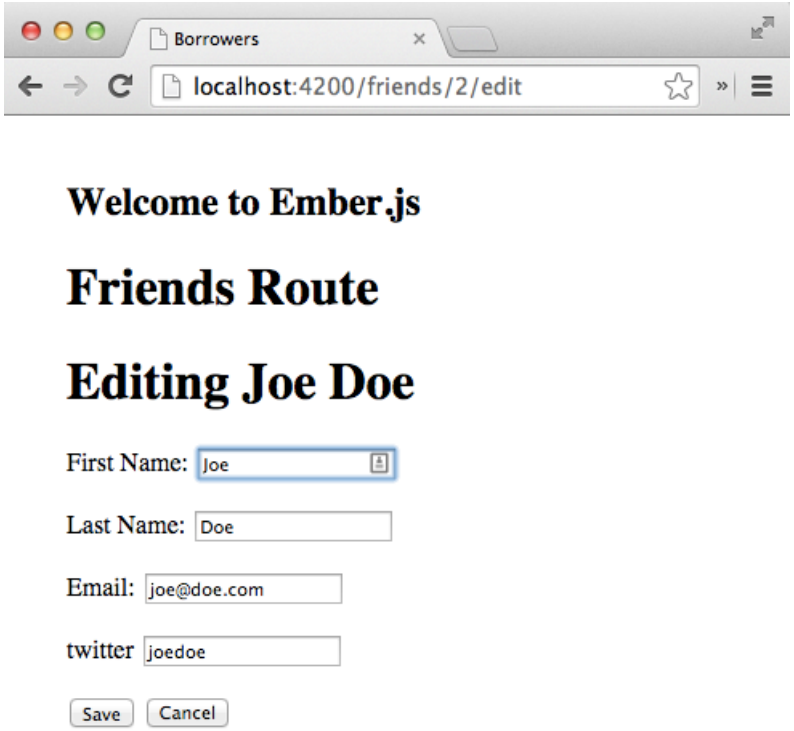
Since the route's path follows the pattern **model_name_id**, we don't need to specify a model hook.

Then we should modify the template **app/templates/friends/edit.hbs** to render the friend's form:

app/templates/friends/edit.hbs

```
<h1>Editing {{model.fullName}}</h1>
{{partial 'friends/form'}}
```

With that in place, let's go to a friend's profile and then append **/edit** in the browser (e.g., http://localhost:4200/friends/2/edit.)



Friends Edit

Thanks to the partial, we have the same form as in the **new template** without writing anything extra. If we open the browser's console and click on **Save** and **Cancel**, we'll see that nothing is handling those actions in the **Friend Edit Controller** and that they are bubbling up the hierarchy chain.

Let's now implement those actions. The **save** action will behave exactly as the one in **new**. We'll do the validations and then, when it has saved successfully, redirect to the profile page. **cancel** will be different; instead of redirecting to the **Friends Index Route**, we'll redirect back to the profile page.

We'll create the controller using **ember g controller**.

```
$ ember g controller friends/edit
version: 0.1.9
installing
  create app/controllers/friends/edit.js
installing
create tests/unit/controllers/friends/edit-test.js
```

Then we can write the same computed property to check whether the object is valid, as well as to check the save and cancel actions.

Write the following in **app/controllers/friends/edit.js**:

app/controllers/friends/edit.js

```
import Ember from 'ember';

export default Ember.Controller.extend({
  isValid: Ember.computed(
    'model.email',
    'model.firstName',
    'model.lastName',
    'model.twitter',
    function() {
      return !Ember.isEmpty(this.get('model.email')) &&
        !Ember.isEmpty(this.get('model.firstName')) &&
        !Ember.isEmpty(this.get('model.lastName')) &&
        !Ember.isEmpty(this.get('model.twitter'));
    }
  ),
  actions: {
    save: function() {
      if (this.get('isValid')) {
        var _this = this;
        this.get('model').save().then(function(friend) {
          _this.transitionToRoute('friends.show', friend);
        });
      } else {
        this.set('errorMessage', 'You have to fill all the fields');
      }
      return false;
    },
    cancel: function() {
      this.transitionToRoute('friends.show', this.get('model'));
      return false;
    }
  }
});
```

If we refresh our browser, edit the profile, and click save, we'll see our changes applied successfully! We can also check that clicking **cancel** takes us back to the user's profile.

To transition from a controller, we have been using **this.transitionToRoute**. It's a helper that behaves similarly to the **{{link-to}}** helper but from within a controller. If we were in a **Route**, we could have used **this.transitionTo**.

Refactoring

Both our **Friends New Controller** and **Friends Edit Controller** share pretty much the same implementation. Let's refactor that creating a base class from which both will inherit.

The only thing that will be different is the **cancel** action. Let's create our base class and then override in every controller according to our needs.

Create a base controller:

```
$ ember g controller friends/base
version: 0.1.9
installing
  create app/controllers/friends/base.js
installing
  create tests/unit/controllers/friends/base-test.js
```

And put the following content in it

app/controllers/friends/base.js

```
import Ember from 'ember';

export default Ember.Controller.extend({
  isValid: Ember.computed(
    'model.email',
    'model.firstName',
    'model.lastName',
    'twitter',
    function() {
      return !Ember.isEmpty(this.get('model.email')) &&
        !Ember.isEmpty(this.get('model.firstName')) &&
        !Ember.isEmpty(this.get('model.lastName')) &&
        !Ember.isEmpty(this.get('model.twitter'));
    }
  ),
  actions: {
    save: function() {
      if (this.get('isValid')) {
        var _this = this;
```

```
        this.get('model').save().then(function(friend) {
          _this.transitionToRoute('friends.show', friend);
        });
      } else {
        this.set('errorMessage', 'You have to fill all the fields');
      }

      return false;
    },
    cancel: function() {
      return true;
    }
  }
});
```

We left **isValid** and **save** exactly as they were, but we have no implementation in the **cancel** action (we just let it bubble up).

We can now replace **app/controllers/friends/new.js** to inherit from **base** and override the cancel action:

app/controllers/friends/new.js

```
import FriendsBaseController from './base';

export default FriendsBaseController.extend({
  actions: {
    cancel: function() {
      this.transitionToRoute('friends.index');
      return false;
    }
  }
});
```

And **app/controllers/friends/edit.js** with:

app/controllers/friends/edit.js

```
import FriendsBaseController from './base';

export default FriendsBaseController.extend({
  actions: {
    cancel: function() {
      this.transitionToRoute('friends.show', this.get('model'));
      return false;
    }
  }
});
```

If we don't override the action, Ember will use the one specified in the base class.

Visiting the edit page.

We can edit a friend now, but we need a way to reach the **edit** screen from the **user profile page**. To do that, we should add a **{{link-to}}** in our **app/templates/friends/show.hbs**.

app/templates/friends/show.hbs

```
<ul>
  <li>First Name: {{model.firstName}}</li>
  <li>Last Name: {{model.lastName}}</li>
  <li>Email: {{model.email}}</li>
  <li>twitter: {{model.twitter}}</li>
  <li>{{link-to "Edit info" "friends.edit" model}}</li>
</ul>
```

If we go to a friend's profile and click **Edit info**, we'll be taken to the edit screen page.

To see all the changes related to this section, refer to the following commit on the project repository Allow to update profiles[47].

[47] https://github.com/abuiles/borrowers/commit/79601014b1567e0ef5c2fda2cd300f3483fa6b22

Deleting friends

We have decided not to lend anything to a couple of friends ever again after they took our beloved **The Dark Side of the Moon** vinyl and returned it with scratches.

It's time to add support to delete some friends from our application. We want to be able to delete them directly within their profile page or when looking at the index.

By now it should be clear how we will do this. Let's use actions.

Our destroy actions will call model#destroyRecord()[48] and then **this.transitionTo** to the **Friends Index Route**.

Let's replace our **app/templates/friends/index.hbs** so it includes the delete action:

app/templates/friends/index.hbs

```
<h1>Friends Index</h1>

<h2>Friends: {{model.length}}</h2>

<table>
  <thead>
    <tr>
      <th>Name</th>
      <th></th>
    </tr>
  </thead>
  <tbody>
    {{#each friend in model}}
      <tr>
        <td>{{link-to friend.fullName "friends.show" friend}}</td>
        <td><a href="#" {{action "delete" friend}}>Delete</a></td>
      </tr>
    {{/each}}
  </tbody>
</table>
```

And then add the action **delete**. This time let's put the delete action on the route **app/routes/friends/index.js**:

[48] http://emberjs.com/api/data/classes/DS.Model.html#method_destroyRecord

app/routes/friends/index.js

```
import Ember from 'ember';

export default Ember.Route.extend({
  model: function() {
    return this.store.findAll('friend');
  },
  actions: {
    delete: function(friend) {
      friend.destroyRecord();
      return false;
    }
  }
});
```

To support deletion on **Friends Show Route**, we just need to add the same link with the action delete and implement the action. Again, we'll put it in the route's actions. In this case, **app/routes/friends/show.js**:

app/routes/friends/show.js

```
import Ember from 'ember';

export default Ember.Route.extend({
  actions: {
    delete: function(friend) {
      var _this = this;

      friend.destroyRecord().then(function() {
        _this.transitionTo('friends.index');
      });
    }
  }
});
```

With that we can now create, update, edit, and delete any of our friends!

Refactoring Time

If we check what we just did, we'll notice that both delete actions are identical except that the one in the index doesn't need to transition since it is already there.

For this specific scenario, calling **this.transitionTo('friends.index')** from within the **Friends Index Route** will behave like a no-op. This is important to mention because we could have one single implementation for the delete action and access it via event bubbling.

We can put the delete action in **app/routes/friends.js**, which is the parent route for both **Friends Index Route** and **Friends New Route**:

app/routes/friends.js

```
import Ember from 'ember';

export default Ember.Route.extend({
  actions: {
    save: function() {
      console.log('save action bubbled to friends route');

      return true;
    },
    cancel: function() {
      console.log('cancel action bubbled to friends route');

      return true;
    },
    delete: function(friend) {
      var _this = this;

      friend.destroyRecord().then(function() {
        _this.transitionTo('friends.index');
      });
    }
  }
});
```

And delete both actions from **app/routes/friends/index.js** and **app/routes/friends/show.js**.

app/routes/friends/index.js

```
import Ember from 'ember';

export default Ember.Route.extend({
  model: function() {
    return this.store.findAll('friend');
  }
});
```

app/routes/friends/show.js

```
import Ember from 'ember';

export default Ember.Route.extend({});
```

Let's breathe slowly and take a moment to enjoy that fresh feeling of deleting repeated code…

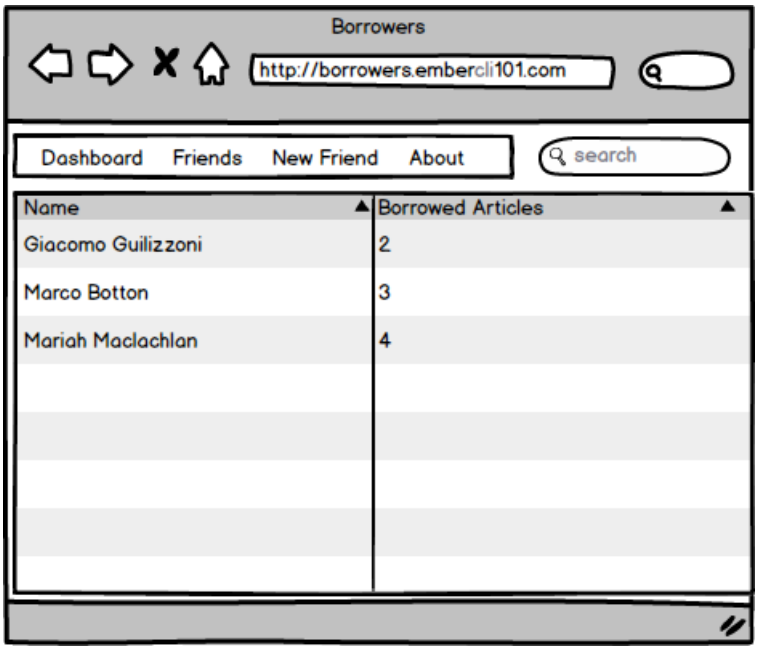
Done?

Next, let's add some styling to our project. We don't want to show this to our friends as it is right now.

Mockups

Before changing our templates, we'll review a couple of mockups to have an idea of how our pages are going to look.

Friends Index

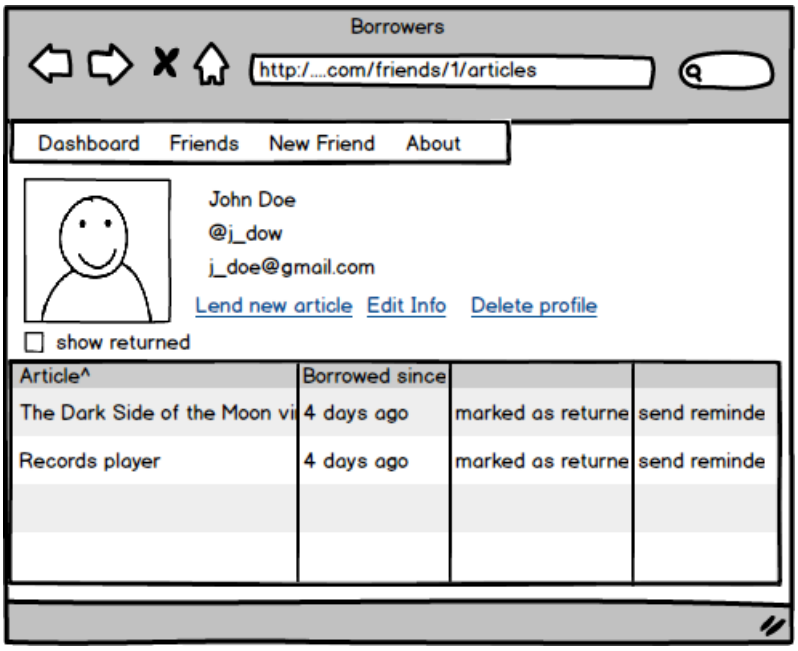


Friends Index

We'll have a header that will take us to a dashboard, the friends index page, and about page. Additionally, we can insert some content depending on which route we are visiting. In the **Friends Index Route** we'll see a search box to filter users.

Then we'll have a table that can be ordered alphabetically or by number of items.

Friend Profile



Friend Profile

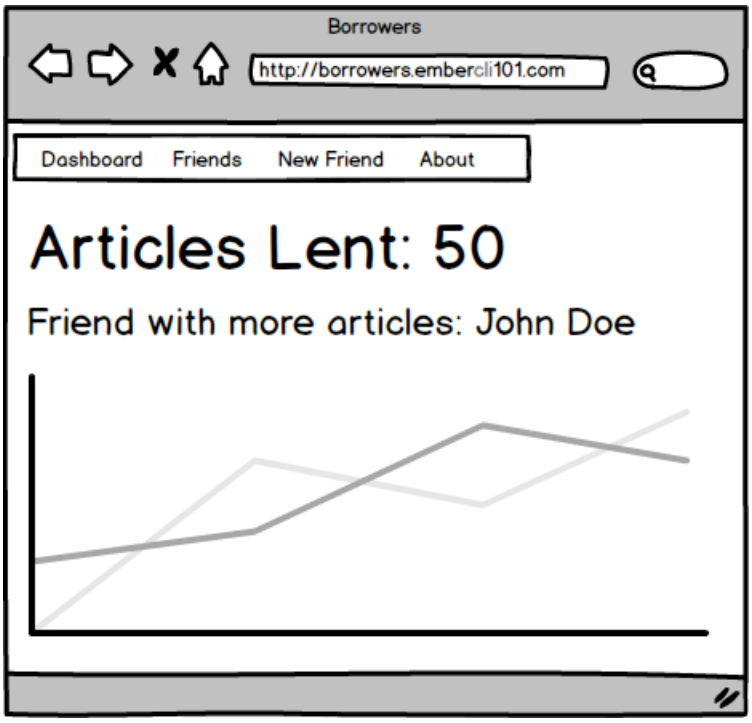
Our friend profile will show us the user's data with an avatar that we might pull from Gravatar.

We have links to add new articles, edit the user's info, or delete the user's profile.

At the bottom we'll have the list of all the articles the user has borrowed with options to mark them as returned or to send a reminder.

If we are careful, we'll also notice that the URL looks a little different from what we currently have. After the friend **id**, we see **/articles** (**..com/friends/1/articles**). Whenever we visit the user profile, the nested resource articles will be rendered by default. We haven't talked about it yet, but basically we are rendering a resource under our **Friends Show Route** that will defer all responsibility of managing state, handling actions, etc. to a different **Controller** and **Route**.

Dashboard



Dashboard

The third mockup is a dashboard where we can ask questions like, "how many articles have we lent to our friends" and "who's the friend with the most articles?" We can also see the number of articles borrowed per day.

Installing Dependencies

To save time, we'll be using picnicss[49] as our base CSS and **fontello** for icons.

Including picnicss

Since picnicss is a front-end dependency, we can use **Bower** to manage such a dependency for us.

First we need to include the following in the file **bower.json**:

[49] http://picnicss.com

Adding picnic to bower.json

```
"picnic": "~3.3.1"
```

Next run **bower install**. Once it is finished, we'll find the picnic assets under **bower_components/picnic/**.

The fact that they are there doesn't mean that they'll be included in our assets. We still need to tell **ember-cli** that we want to **import** those assets into our application. To do so, we need to add the following line to our Brocfile.js before `module.exports = app.toTree();`

Adding picnic to the Brocfile

```
/* global require, module */

var EmberApp = require('ember-cli/lib/broccoli/ember-app');

var app = new EmberApp();

app.import('bower_components/picnic/releases/picnic.min.css');

module.exports = app.toTree();
```

app.import is a helper function that tells **ember-cli** to append **bower_components/picnic/releases/picninc.min.css** into our assets. By default it will put any **CSS** file we import into **/vendor.css** and any JavaScript file into **/vendor.js**.

If we check **app/index.html**, we'll see 2 CSS files included:

app/index.html

```
<link rel="stylesheet" href="assets/vendor.css">
<link rel="stylesheet" href="assets/borrowers.css">
```

The first one contains all the imported (vendor) **CSS files** and the second one contains the **CSS files** we defined under **app/styles**.

Why have two separate CSS and JavaScript files? Vendor files are less likely to change, so we can take advantage of caching when we deploy our application. While our app CSS and JS might change, vendor files will stay the same, allowing us to take advantage of the cache.

After modifying our `Brocfile` we need to stop and start the server again so the changes are applied. Once we have done that, we can we refresh our browser and go to **http://localhost:4200/assets/vendor.css**, we'll see that the code for **picnicss** is there.

Including fontello

Because fontello[50] doesn't have a custom distribution we can download with **bower**, we'll download a bundle of icons and fonts that we can manage manually by putting it under **vendor/fontello**.

With bower dependencies, we don't have to worry about keeping things under our revision control system because bower will take care of downloading them for us. Howevever, we do have to keep track of dependencies not managed by bower.

We can download a bundle from the following URL http://cl.ly/3y1W1B3Y4028 and then put the content under **vendor/**, which will give us the directory **vendor/fontello**.

In order to tell **ember-cli** that we want to include fontello's CSS and fonts, we need to modify our Brocfile as follows:

Brocfile.js

```
var EmberApp = require('ember-cli/lib/broccoli/ember-app');

var app = new EmberApp();

app.import('vendor/fontello/fontello.css');
app.import('vendor/fontello/font/fontello.ttf', {
  destDir: 'font'
});
app.import('vendor/fontello/font/fontello.eot', {
  destDir: 'font'
});
app.import('vendor/fontello/font/fontello.svg', {
  destDir: 'font'
});
app.import('vendor/fontello/font/fontello.woff', {
  destDir: 'font'
});
```

We are already familiar with the line to import **fontello.css**, but the following ones are new to us since we have never passed any option to **import**.

The option **destDir** tells **ember-cli** that we want to put those files under a directory called **font**. If we save and refresh our browser, **vendor.css** should now include **fontello.css**. We can also check the files in **font** by going to http://localhost:4200/font.

[50] http://fontello.com/

Check the change on GitHub by visiting the following commit: Add fontello and picnicss[51].

With that, we know the basics of including vendor files. Now that we have our basic dependencies on hand, let's improve the appearance of our templates.

The header

We'll use partials as much as possible to simplify our templates. In this case, we'll create a partial that contains the code for the navigation bar. Create the file **app/templates/partials/-header.hbs** with the following content:

app/templates/partials/-header.hbs

```
<nav>
  {{link-to "Borrowers" "index" class="main"}}

  <!-- responsive -->
  <input id="bmenu" class="burgercheck" type="checkbox">
  <label for="bmenu" class="burgermenu"></label>
  <!-- /responsive -->

  <div class="menu">
    {{link-to "Dashboard" "index" class="icon-gauge"}}
    {{link-to "Friends" "friends" class="icon-users-1"}}
    {{link-to "New Friend" "friends.new" class="icon-user-add"}}
  </div>
</nav>
```

The header should always be visible in our application. In Ember, the right receptacle for that content would be the **Application Template** since it will contain any other template inside its **{{outlet}}**.

Modify **app/templates/application.hbs** as follows:

[51] https://github.com/abuiles/borrowers/commit/90a1ea3fe6320ad1746b4c0ab4069401d2fd6247

app/templates/application.hbs

```
{{partial 'partials/header'}}

<div class="row">
  <div class="full">
    {{outlet}}
  </div>
</div>
```

We will render the header and wrap the outlet in a row using **picnicss** classes.

If we refresh, the header should display nicely.

Friends Index

First, let's remove the **<h1>** from **app/templates/friends.hbs** so it only contains **{{outlet}}**. Next, clean up **app/templates/friends/index.hbs** so it adds the class **primary** to the table:

app/templates/friends/index.hbs

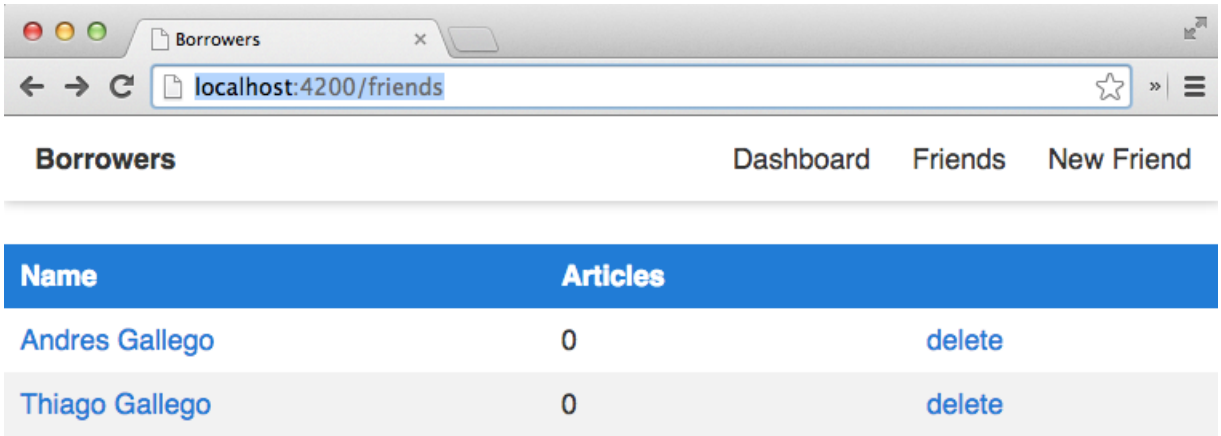
```
<table class="primary">
  <thead>
    <tr>
      <th>Name</th>
      <th>Articles</th>
      <th></th>
    </tr>
  </thead>
  <tbody>
    {{#each friend in model}}
      <tr>
        <td>{{link-to friend.fullName "friends.show" friend}}</td>
        <td>{{friend.totalArticles}}</td>
        <td><a href="#" {{action "delete" friend}}>delete</a></td>
      </tr>
    {{/each}}
  </tbody>
</table>
```

Then we need to add some extra styling to the table. We want it to be full width, so let's modify **app/styles/app.css** as follows:

app/styles/app.css

```
body {
  display: block;
  text-align: center;
  color: #333;
  background: #FFF;
  margin: 80px auto;
  width: 100%;
}

table {
  width: 100%;
}
```

Now if we visit http://localhost:4200/friends, we should see:



Friends Index

New Friend And Friend profile template

Next let's modify **app/templates/friends/-form.hbs**

app/templates/friends/-form.hbs

```
<form {{action "save" on="submit"}}>
  <h2>{{errorMessage}}</h2>
  <fieldset>
    {{input value=model.firstName placeholder="First Name"}}<br>
    {{input value=model.lastName  placeholder="Last Name"}}<br>
    {{input value=model.email     placeholder="email"}}<br>
    {{input value=model.twitter   placeholder="twitter"}}<br>
    <input type="submit" value="Save" class="primary">
    <button {{action "cancel"}}>Cancel</button>
  </fieldset>
</form>
```

And finally, change **app/templates/friends/show.hbs**.

app/templates/friends/show.hbs

```
<div class="friend-profile">
  <p>{{model.firstName}}</p>
  <p>{{model.lastName}}</p>
  <p>{{model.email}}</p>
  <p>{{model.twitter}}</p>
  <p>{{link-to "Edit info" "friends.edit" model}}</p>
  <p><a href="#" {{action "delete" model}}>delete</a></p>
</div>
```

The Dashboard

By default, we'll use the **Application Index Route** as the dashboard. For now, we are going to create the file **app/templates/index.hbs** and write **<h2>Dashboard</h2>**.

Let's move on with more functionality.

Articles Resource

With our **Friends** CRUD ready, we can start lending articles.

Let's create an articles resource:

```
$ ember generate resource articles createdAt:date description:string notes:strin\
g state:string
  create app/models/article.js
  create tests/unit/models/article-test.js
  create app/routes/articles.js
  create app/templates/articles.hbs
  create tests/unit/routes/articles-test.js
```

Let's check the model.

app/models/article.js

```
import DS from 'ember-data';

export default DS.Model.extend({
  createdAt: DS.attr('date'),
  description: DS.attr('string'),
  notes: DS.attr('string'),
  state: DS.attr('string')
});
```

We have defined our **Articles** model successfully, but we need to wire the relationship between **Friends** and **Articles**. Let's do that next.

Defining relationships.

We have to specify that a friend can have many articles and that those articles belong to a friend. In other frameworks this is known as **hasMany** and **belongsTo** relationships, and so they are in Ember-Data.

Remember, Ember doesn't include data handling support by default. This is accomplished through Ember-Data, which is the official library for this.

If we want to add a **hasMany** relationship to our models, we write:

```
  articles: DS.hasMany('article')
```

Or we want a **belongsTo**:

```
  friend: DS.belongsTo('friend')
```

Using the previous relationship types, we can modify our **Article** model:

app/models/article.js

```
import DS from 'ember-data';

export default DS.Model.extend({
  createdAt:   DS.attr('date'),
  description: DS.attr('string'),
  friend:      DS.belongsTo('friend'),
  notes:       DS.attr('string'),
  state:       DS.attr('string')
});
```

And our **Friend** model to add the **hasMany** to articles:

app/models/friend.js

```
import DS from 'ember-data';
import Ember from 'ember';

export default DS.Model.extend({
  articles:      DS.hasMany('article'),
  email:         DS.attr('string'),
  firstName:     DS.attr('string'),
  lastName:      DS.attr('string'),
  totalArticles: DS.attr('number'),
  twitter:       DS.attr('string'),
  fullName: Ember.computed('firstName', 'lastName', function() {
    return this.get('firstName') + ' ' + this.get('lastName');
  })
});
```

With just those two lines, we have added a relationship between our models. Now let's work on the **Articles** resource.

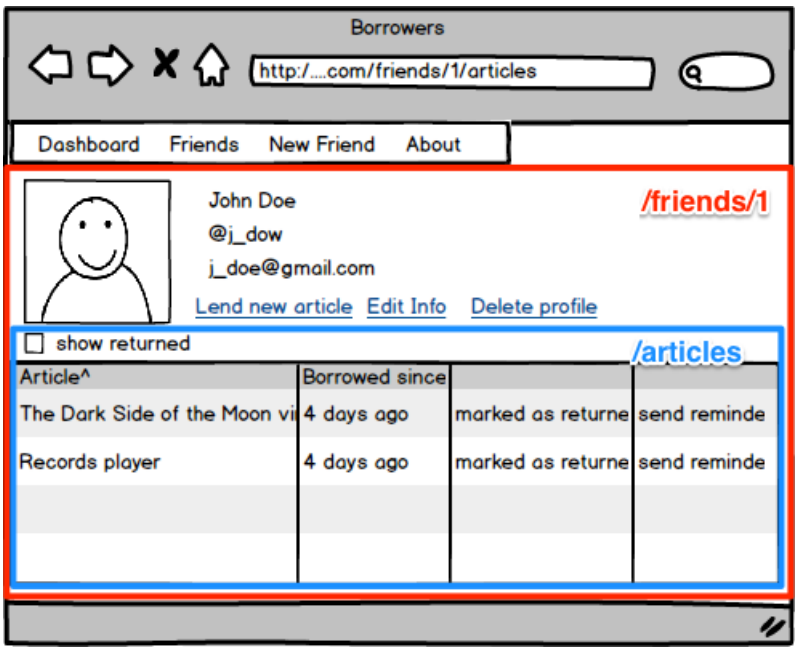
Specifying relationships with the generator.

We can add `hasMany` or `belongsTo` relationships when running the generator, we didn't use it when we created the articles resource so we could explain relationships, but we could have done the following: `ember g resource articles friend:belongsTo ...`.

Nested Articles Index

In our **Friend Profile** mockup, we specified that we wanted to render the list of articles as a nested route inside the friend profile.

If we look again at the mockup now highlighting the nested routes,



Friend Profile with nested routes

the part in red corresponds to the **Friends Show Route** while the part in blue is where all routes belonging to the resource **Articles** will go.

We need to make a couple of changes to handle this scenario. First we need to make sure that **articles** is specified as a nested resource inside **Friends Show**. Let's go to our **app/router.js** and change it to reflect this:

app/router.js

```
this.resource('friends', function(){
  this.route('new');
  this.route('show', { path: ':friend_id' }, function() {
    this.resource('articles', function() { });
  });
  this.route('edit', { path: ':friend_id/edit' });
});

export default Router;
```

Now let's open the **ember-inspector** and check our newly defined routes:

friends.show	FriendsShowRoute	>$E	FriendsShowController	friends/show	
friends.show.loading	FriendsShow.LoadingRoute	>$E	FriendsShow.LoadingController	friends/show.loading	/friends/:friend_id/loading
friends.show.error	FriendsShow.ErrorRoute	>$E	FriendsShow.ErrorController	friends/show.error	
articles	ArticlesRoute	>$E	ArticlesController	articles	
articles.loading	ArticlesLoadingRoute	>$E	ArticlesLoadingController	articles/loading	/friends/:friend_id/articles/loading
articles.error	ArticlesErrorRoute	>$E	ArticlesErrorController	articles/error	
articles.index	ArticlesIndexRoute	>$E	ArticlesIndexController	articles/index	/friends/:friend_id/articles

Nested Articles Routes

We can identify the routes and controllers that Ember expects us to define for the new resource.

Next we need to add an **{{outlet }}** to **app/templates/friends/show.hbs**, which is where the nested routes will render:

app/templates/friends/show.hbs

```
<div class="friend-profile">
  <p>{{model.firstName}}</p>
  <p>{{model.lastName}}</p>
  <p>{{model.email}}</p>
  <p>{{model.twitter}}</p>
  <p>{{link-to "Edit info" "friends.edit"  model}}</p>
  <p><a href="#" {{action "delete" model}}>delete</a></p>
</div>
<div class="articles-container">
  {{outlet}}
</div>
```

Any nested route or resource will be rendered by default into its parent's **{{outlet}}**.

Rendering the index.

Let's create a new file called **app/templates/articles/index.hbs** and write the following:

app/templates/articles/index.hbs

```
<h2>Articles Index</h2>
```

If we visit a friend profile, we won't see anything related with the **Articles Index Route**. Why? Well, we are not visiting that route, that's why. To get to the **Articles Index Route**, we need to modify the **link-to** in **app/templates/friends/index.hbs** to reference the route **articles** instead of **friends.show**. We'll still pass the **friend** as an argument since the route **articles** is nested under **friends.show** and it has the dynamic segment **:friend_id**.

app/templates/friends/index.hbs

```
<td>{{link-to friend.fullName "articles" friend}}</td>
```

Now, with the previous change, if we go to the friends index and visit any profile, we'll see **Articles Index** at the bottom.

Opening the **ember-inspector** and filtering by *Current Route only**, we'll see:

Current Route only

Route Name	Route
application	ApplicationRoute
friends	FriendsRoute
friends.show	FriendsShowRoute
articles	ArticlesRoute
articles.index	ArticlesIndexRoute

Articles Index Route

Routes are resolved from top to bottom, so when we navigate to **/friends/1/articles** it will go first to the **ApplicationRoute** and move to **FriendsShowRoute** to fetch our friend. Once it is loaded, it will move to **ArticlesIndexRoute**.

Next we need to define the model hook for the **ArticlesIndexRoute**.

Fetching our friend articles.

Let's add the **Articles Index Route** to the generator and reply 'no' when it asks us if we want to overwrite the template.

```
$ ember g route articles/index
version: 0.1.9
installing
[?] Overwrite /borrowers/app/templates/articles/index.hbs? (Yndh) n

Overwrite /borrowers/app/templates/articles/index.hbs? No, skip
create app/routes/articles/index.js
skip app/templates/articles/index.hbs
installing
  create tests/unit/routes/articles/index-test.js
```

In **app/routes/articles/index.js**, load the data using the model hook:

app/routes/articles/index.js

```
import Ember from 'ember';

export default Ember.Route.extend({
  model: function() {
    return this.modelFor('friends/show').get('articles');
  }
});
```

In the model hook, we are using a new function this.modelFor[52] that helps us grab the model for any parent route. In this scenario, parent routes are all the ones appearing on top of **ArticlesIndexRoute** in the **ember-inspector**.

Once we get the model for **FriendsShowRoute**, we simply ask for its articles. And that's what we are returning.

We need to modify the **app/templates/articles/index.hbs** so it displays the articles:

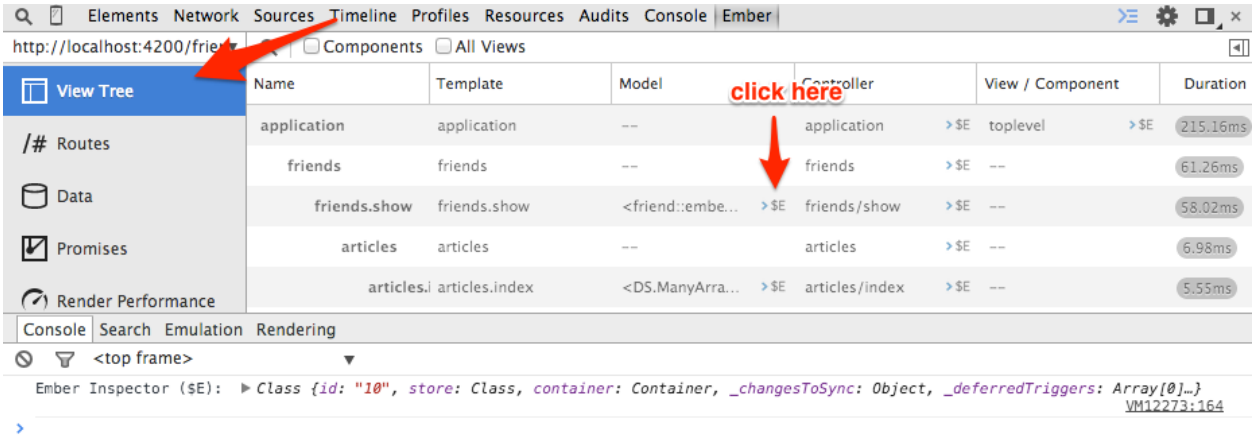
app/templates/articles/index.hbs

```
<table class="primary">
  <thead>
    <tr>
      <th>Description</th>
      <th>Borrowed since</th>
      <th></th>
      <th></th>
    </tr>
  </thead>
  <tbody>
    {{#each article in model}}
      <tr>
        <td>{{article.description}}</td>
        <td>{{article.createdAt}}</td>
        <td></td>
        <td></td>
      </tr>
    {{/each}}
  </tbody>
</table>
```

[52] http://emberjs.com/api/classes/Ember.Route.html#method_modelFor

If our friend doesn't have articles yet, we can use the **ember-inspector** to add some manually.

Let's open the **ember-inspector** and select the model from the route *friends.show**:



Select Friend Model

Once we have the instance of a friend assigned to the variable **$E**, let's run the following on the browser's console:

```
$E.get('articles').createRecord({description: 'foo'})
$E.get('articles').createRecord({description: 'bar'})
```

We will notice that our Friend Index updates automatically with the records we create.

So far we are only putting records into the store, but they are not being saved to our backend. To do that we'll need to call **save()** on every instance. Let's try to call save:

```
$E.get('articles').createRecord({description: 'foo'}).save()
```

We will notice that a **POST** is attempted to our backend, but it gets rejected because the model is not valid:

```
Error: The backend rejected the commit because it was invalid: {state: can't be \
blank,is not included in the list}
```

Let's add the route **Articles New** and the template so we can lend new articles to our friends.

Check the following commit to review all the changes of the previous chapter: Add articles index[53]

[53] https://github.com/abuiles/borrowers/commit/4346a795210ba3d46d02952611f0b91f9f140434

Sideloading Articles

If we visit http://api.ember-cli-101.com/api/friends[54], we'll notice that there is no information about any of our friends' articles. We omit that information intentionally so the early version of the application won't break.

However, from now on, we need to include the articles so that they are displayed when we visit a friend's profile. To accomplish this we'll use version 2 (V2) of the borrowers backend API, which includes the articles for every user. We can try it out by visiting http://api.ember-cli-101.com/api/v2/friends[55].

How do we use the new version of the API? We need to modify the property **namespace** in the application adapter so it refers to **api/v2**. Let's change **app/adapters/application.js** to look like the following:

app/adapters/application.js

```
import DS from 'ember-data';

export default DS.ActiveModelAdapter.extend({
  namespace: 'api/v2'
});
```

Once we have made that change, we'll consume the new version of the API.

Sideloading data is one of the different strategies we have in Ember-Data to work with relationships. We'll explore other alternatives in a later chapter dedicated to Ember-Data.

Lending new articles

Let's start by adding the route. We've done it with the generator up to this point, but now we'll do it manually.

We need to add the nested route **new** under the resource **articles**:

[54] http://api.ember-cli-101.com/api/friends
[55] http://api.ember-cli-101.com/api/v2/friends

app/router.js

```
import Ember from 'ember';
import config from './config/environment';

var Router = Ember.Router.extend({
  location: config.locationType
});

Router.map(function() {
  this.resource('friends', function() {
    this.route('new');
    this.route('show', { path: ':friend_id' }, function() {
      this.resource('articles', function() {
        this.route('new');
      });
    });
    this.route('edit', { path: ':friend_id/edit' });
  });
});

export default Router;
```

Then let's create the route **app/routes/articles/new.js** with the model hook and actions support:

app/routes/articles/new.js

```
import Ember from 'ember';

export default Ember.Route.extend({
  model: function() {
  return this.store.createRecord('article', {
      state: 'borrowed',
      friend: this.modelFor('friends/show')
    });
  },
  actions: {
    save: function() {
      var _this = this;
      var model = this.modelFor('articles/new');

      model.save().then(function(){
        _this.transitionTo('articles');
```

```
      });
    },
    cancel: function() {
      this.transitionTo('articles');
    }
  }
});
```

In the model hook we use this.store.createRecord[56], which creates a new instance of a model in the store. It takes the name of the model we're creating and its properties.

We pass the property **friend** and **state**. The former will make sure that the article is linked with our friend, and the latter is simply setting the state attribute. We'll start it in **borrowed**.

Ember-Data allows us to specify a **defaultValue** for our attributes. We can use that instead of doing it explicitly in the model hook. In **app/models/article.js**, let's replace the definition of **state** so it looks as follows:

app/models/article.js

```
  state: DS.attr('string', {
    defaultValue: 'borrowed'
  })
```

Then we can modify our model in **app/routes/articles/new.js** so it doesn't add the initial state:

app/routes/articles/new.js

```
  model: function() {
    return this.store.createRecord('article', {
      friend: this.modelFor('friends/show')
    });
  },
```

In our friends example we put the **save** and **cancel** actions in the controller, but this time we are defining it in the route. The question is: where do we need to put this kind of action?

We used both strategies as an example that we can get to the same results using either the route or controller. However, the rule of thumb is that we keep every action that modifies our application state in the routes and use the controllers as decorators for our templates. Actions like saving, destroying, and creating new objects are best fit for the route.

[56] http://emberjs.com/api/data/classes/DS.Store.html#method_createRecord

Common patterns on resource routes model hooks

- Edit and Show Route: **return this.store.find('modelName', modelId)**
- Create Route: **return this.store.createRecord('modelName', properties)**
- Index Route: **return this.store.findAll('modelName')**

Next we need to add the **new** template. Since we might want to reuse the **form**, let's add it in a partial and then include it in the template **app/templates/articles/new.hbs**.

We'll create the **-form** partial in **app/templates/articles/-form.hbs**. Remember, partial names begin with a dash:

app/templates/articles/-form.hbs

```
<form>
  <h2>{{errorMessage}}</h2>
  <fieldset>
    {{input value=model.description placeholder="Description"}}<br>
    {{input value=model.notes  placeholder="Notes"}}<br>
    <button {{action "save"}} class="primary">Save</button>
    <button {{action "cancel"}}>Cancel</button>
  </fieldset>
</form>
```

Then include it in **app/templates/articles/new.hbs**:

app/templates/articles/new.hbs

```
<h2> Lending new articles</h2>
{{partial "articles/form"}}
```

We are almost done. We have set up the route and template, but we still haven't added a link to navigate to the **Articles New Route**. Let's add **link-to** to **articles.new** in **app/templates/friends/show.hbs**:

app/templates/friends/show.hbs

```
<div class="friend-profile">
  <p>{{model.firstName}}</p>
  <p>{{model.lastName}}</p>
  <p>{{model.email}}</p>
  <p>{{model.twitter}}</p>
  <p>{{link-to "Lend article" "articles.new"}}</p>
  <p>{{link-to "Edit info" "friends.edit"  model}}</p>
  <p><a href="#" {{action "delete" model}}>delete</a></p>
</div>
<div class="articles-container">
  {{outlet}}
</div>
```

We are creating the link with `{{link-to "Lend articles" "articles.new"}}`. Since we're already in the context of a friend, we don't need to specify the dynamic segment. If we want to add the same link in the **Friends Index Route**, we'll need to pass the parameter as **{{link-to "Lend articles" "articles.new" friend}}** where **friend** is an instance of a **friend**.

Tasks

Create an **Articles New Controller** and validate that the model includes **description**. If it is valid, let the action bubble to the route. Otherwise, set an **errorMessage**.

Click the following link for a list of changes introduced in this chapter: http://git.io/wYEikg[57].

[57] http://git.io/wYEikg

Computed Property Macros

In **app/controllers/friends/base.js**, we define the computed property **isValid** with the following code:

Computed Property isValid is app/controllers/friends/base.js

```
isValid: Ember.computed(
  'model.email',
  'model.firstName',
  'model.lastName',
  'model.twitter',
  function() {
    return !Ember.isEmpty(this.get('model.email')) &&
      !Ember.isEmpty(this.get('model.firstName')) &&
      !Ember.isEmpty(this.get('model.lastName')) &&
      !Ember.isEmpty(this.get('model.twitter'));
  }
),
```

Although the previous code does what we expect, it is not the most pleasant to read, especially with all those nested **&&'s**. As it turns out, Ember has a set of helper functions that will allow us to write the previous code in a more idiomatic way using something called computed property macros.

Computed property macros are a set of functions living under **Ember.computed**. that allow us to create computed properties in an easier, more readable and clean way.

As an example, let's take two computed property macros and write our **isValid** on terms of them:

- Ember.computed.and[58]
- Ember.computed.notEmpty[59]

[58] http://emberjs.com/api/#method_computed_and
[59] http://emberjs.com/api/#method_computed_notEmpty

Computed Property With Macros in app/controllers/friends/base.js

```
export default Ember.Controller.extend({
  hasEmail:     Ember.computed.notEmpty('model.email'),
  hasFirstName: Ember.computed.notEmpty('model.firstName'),
  hasLastName:  Ember.computed.notEmpty('model.lastName'),
  hasTwitter:   Ember.computed.notEmpty('model.twitter'),
  isValid:      Ember.computed.and(
    'hasEmail',
    'hasFirstName',
    'hasLastName',
    'hasTwitter'
  ),

// actions omitted
```

This is certainly much cleaner and less error-prone.

We can see the full list of computed properties with Ember.computed.alias[60].

Using components to mark an article as returned.

We previously lent our favorite Whisky glass to one of our friends and they just returned it. We need to mark the item as returned.

Our interface will look similar to the following. We can select the state of the article within the articles index. Whenever that article has pending changes, we'll see a **save** button.

Description	Notes	Borrowed since	
Whisky glass		Mon Sep 29 2014 10:12:50 GMT-0500 (COT)	borrowed

Articles Index with Selector

Using components we'll encapsulate the behavior per row into its own class removing responsibility from the model and delegating it to a class that will handle how every row should look and additionally when it should fire a save depending on the state of every article.

We'll create an **articles/article-row** component which will wrap every element. We'll pass the necessary data to render the list of possible states and also the article.

Let's create the **articles/article-row** using the components generator.

[60] http://emberjs.com/api/#method_computed_alias

Creating an component

```
$ ember g component articles/article-row
version: 0.1.9
installing
  create app/components/articles/article-row.js
  create app/templates/components/articles/article-row.hbs
installing
  create tests/unit/components/articles/article-row-test.js
```

Let's modify the component so it looks as follows:

app/components/articles/article-row.js

```
import  Ember from 'ember';

export default Ember.Component.extend({
  tagName: 'tr',
  article: null, // passed-in
  articleStates: null,   // passed-in
  actions: {
    saveArticle: function(article) {
      this.sendAction('save', article);
    }
  }
});
```

We are specifying that the type for this component is going to be a `tr` meaning that whatever content we put in the template, it will be wrapped in table row using the HTML tag **tr**, by default it is a **div**. Also we defined two properties `articles` and `articleStates` with value `null` and the comment: "passed-in". It will help people consuming the component to identify which data they should pass-in.

We also added an action "saveArticle" which receives an article and then calls `this.sendAction` with the arguments `save` and the `article`.

Unlike controllers, actions in components won't bubble up automatically, so if we want to call an action in a Route or Controller from a component we'll need to bind such action to a property and then call it using `sendAction`, we'll see shortly how that look in a template.

We need to add the the markup for the component as follows:

app/templates/components/articles/article-row.hbs

```
<td>{{article.description}}</td>
<td>{{article.notes}}</td>
<td>{{article.createdAt}}</td>
<td>{{view "select" content=articleStates selection=article.state}}</td>
<td>
  {{#if article.isSaving}}
    <p>Saving ...</p>
  {{else if article.isDirty}}
    <button {{action "saveArticle" article}}>Save</button>
  {{/if}}
</td>
```

In the template we are defining the cells for every article row and reading the value from the "passed-in" property article, notice also that we are calling the action "saveArticle" not "save".

We are also using the Ember.Select[61] view, which is a helper that allows us to render a HTML **select** element and bind the value to a given property.

```
<td>{{view "select" content=articleStates selection=article.state}}</td>
```

We pass **content**, which contains available options, and we specify which attribute will be bound through the attribute **selection**.

If we were passing a collection of objects, then we would have to specify the properties **optionValuePath** and **optionLabelPath**.

We are also using the properties **article.isSaving** and **article.isDirty**, which belong to the article we passed-in.

The previous properties are part of DS.Model[62] and they help us to know things about a model. In the previous scenario, **article.isDirty** becomes true if there is a change to the model and **article.isSaving** is true if the model tries to persist any changes to the backend.

Before using our components, let's add to our articles index controller a property called `possibleStates` which we'll use to pass down to the component:

[61] http://emberjs.com/api/classes/Ember.Select.html
[62] http://emberjs.com/api/data/classes/DS.Model.html

app/controllers/articles/index.js

```
import Ember from 'ember';

export default Ember.Controller.extend({
  queryParams: ['show'],
  possibleStates: ["borrowed", "returned"],
  // ...
```

We can create the controller with the controller generator, run: `ember g controller articles/index`.

Now let's use our component in the articles index template:

app/templates/articles/index.hbs

```
<table class="primary">
  <thead>
    <tr>
      <th>Description</th>
      <th>Notes</th>
      <th>Borrowed since</th>
      <th></th>
      <th></th>
    </tr>
  </thead>
  <tbody>
    {{#each model as |article|}}
      {{articles/article-row  article=article save="save" articleStates=possible\
States}}
    {{/each}}
  </tbody>
</table>
```

We are iterating over every article in the model and then rendering an articles-row component for each of them, we are passing as attributes the article, bounding the save action to another action which is also called "save" and finally binding the `articleStates` to the list of `possibleStates` in the controller. We could have call both properties the same but we did not, so we could demonstrate that we are only assigning variables, currently is a two-way binding, meaning that if we modify the list in any of the component then it will be modified in the controller too, in future versions will be able to specify a one-way binding meaning that this data will be read-only.

To clarify a bit further, with `save="save"` as soon as we call `this.sendAction('save', article)` then if the controller has an action called save, it will be called otherwise it will keep bubbling as any other action (controller, route, parents and so on).

In upcoming versions of Ember, we'll be able to use components as if they were just another HTML tag, so we could write <articles/article-row> instead of {{articles/article-row}}.

If we open the **ember-inspector**, open the view tree and then select components, we will notice that every component is displayed independently.

is-attributes

The following are the attributes of the type **isSomething** and can be found in DS.Model documentation[63]: * isDeleted * isDirty * isEmpty * isError * isLoaded * isLoading * isNew * isReloading * isSaving * isValid

If we go to the browser and try what we just created, everything should work. Except that if we click save, our object is not saved because we don't have a handler for the **save** action.

We can add one in **app/routes/articles/index.js**:

Add save action to app/routes/articles/index.js

```
import Ember from 'ember';

export default Ember.Route.extend({
  model: function() {
    return this.modelFor('friends/show').get('articles');
  },
  actions: {
    save: function(model) {
      model.save();
      return false;
    }
  }
});
```

Remember that actions always bubble to the parents. If we had a **save** action in the index controller, it would have been called first and then bubbled up if we returned **true**.

[63] http://emberjs.com/api/data/classes/DS.Model.html#property_isDeleted

Implementing auto save.

Instead of clicking the save button every time we change the state of the model, we want it to save automatically.

First we'll rewrite our template so the button part is not included.

app/templates/components/articles/article-row.hbs

```
<td>{{article.description}}</td>
<td>{{article.notes}}</td>
<td>{{article.createdAt}}</td>
<td>{{view "select" content=articleStates selection=article.state}}</td>
<td>
  {{#if article.isSaving}}
    <p>Saving ...</p>
  {{/if}}
</td>
```

On the component, we need to set up an observer on the **state** property and call an **autoSave** function that will fire the action **save**:

app/components/articles/article-row.js

```
import Ember from 'ember';

export default Ember.Component.extend({
  tagName: 'tr',
  article: null, // passed-in
  articleStates: null, // passed-in
  autoSave: function() {
    var article = this.get('article');
    if (!article.get('isNew')) {
      this.sendAction('save', article);
    }
  },
  stateChanged: Ember.observer('article.state', function() {
    var article = this.get('article');
    if (article.get('isDirty') && !article.get('isSaving')) {
      Ember.run.once(this, this.autoSave);
    }
  }).on('init')
});
```

The function **autoSave** is in charge of firing up an action using **this.sendAction**. We want to make sure the record is not in **state** `isNew`:

app/components/articles/article-row.js

```
  autoSave: function() {
    var article = this.get('article');
    if (!article.get('isNew')) {
      this.sendAction('save', article);
    }
  }
```

Then we set up an observer on the **article.state** property. By default, observers are not set up until the function where they are specified is consumed. We pass **on('init')**, which will call the function as soon as the controller is initialized. This helps us activate the observer.

app/components/articles/article-row.js

```
  stateChanged: Ember..observer('article.state', function() {
    var article = this.get('article');
    if (article.get('isDirty') && !article.get('isSaving')) {
      Ember.run.once(this, this.autoSave);
    }
  }).on('init')
```

We check whether the model has pending changes and make sure that it is not currently saving anything. If both conditions are true, we set up a call to **autoSave** using **Ember.run.once(this, this.autoSave)**.

The question now is: what is **Ember.run.once**? We need to emphasize that observers are synchronous. They are called as soon as their observed property changes, so we can have scenarios where the same function is called twice. Let's check the following scenario where we observe **a** and **b** calling an expensive operation when either property changes.

Observer example

```
abChange: Ember.observer('a', 'b', function() {
  this.expensiveOperation();
})
```

Now, if we do something like the following, then an expensive operation will be called twice:

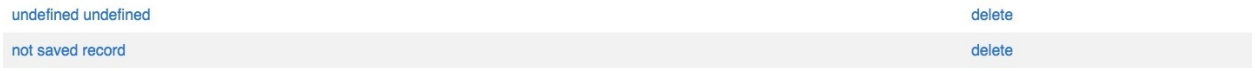
```
this.set('a', 2);
this.set('b', 3);
```

To avoid this situation we use Ember.run.once[64], which guarantees that the function passed will be called only once during the current running loop. If we set **a** and **b** consecutively, the observer functions is still called twice but the expensive operations just once.

Observers require more than what we just covered; they and the run loop will be discussed at greater length in a later chapter.

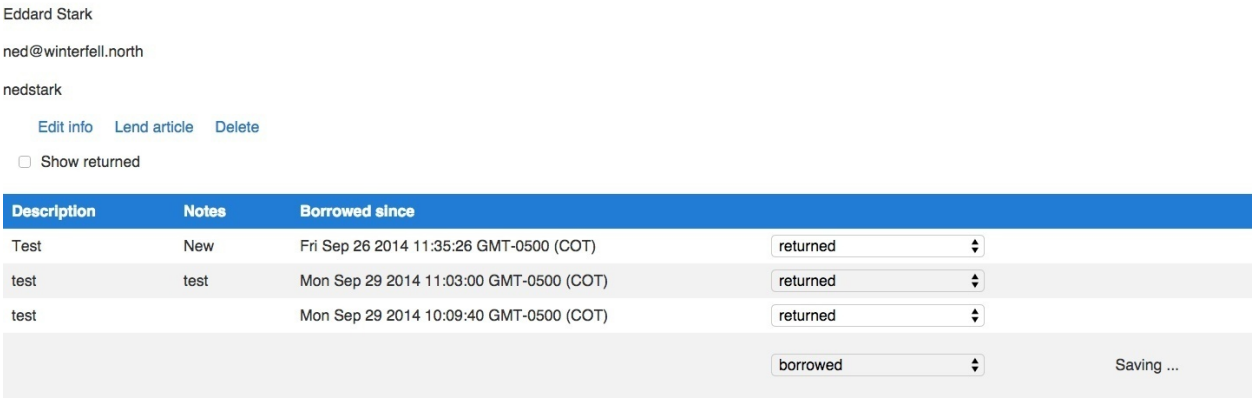
Route hooks

If we go to http://localhost:4200/friends/new[65] and click cancel without entering anything, or we write something and then click cancel, we'll still see the unsaved record in our **Friends Index**. It only goes away if we refresh the app.



Unsaved friends

The same happens with an article. If we try to create one but we click cancel, it will appear in the index anyway.



Unsaved articles

It is important to remember that the **Ember-Data Store** not only keeps all the data we load from the server, but it also keeps the one we create on the client. We were actually pushing a new record to the store when we did the following on the **Friends New Route**:

[64] http://emberjs.com/api/classes/Ember.run.html#method_once
[65] http://localhost:4200/friends/new

```
  model: function() {
    return this.store.createRecord('friend');
  },
```

Such records will live in the store with the state **new**. We can call **save** on it, which will persist it to the backend and make it move to a different state, or we can remove it and our backend will never know about it.

We might ask ourselves: but aren't we doing a **store.findAll** on the **Friends Index Route**, which loads our data again from the server? And shouldn't that remove the unsaved records?

That's partially true. It is correct that when we do **this.store.findAll('friend')**, a **GET** request is made to the server. When we load our existing records again, instead of throwing out all the records in the store, **Ember-Data** merges the results, updating existing records and leaving untouched the ones that the server doesn't know about. That's why we see the new but unsaved record in the index.

To mitigate this situation, if we are leaving the **Friends New Route** and the model was not saved, we'll need to remove the record we created from the store. How do we do that?

Ember.Route[66] has a set of hooks that are called at different times during the route lifetime. For instance, we can use activate[67] to do something when we enter a route, deactivate[68] when we leave it or resetController[69] to reset values on some actions.

Let's try them in **app/routes/friends/new.js**:

Using Route Hooks in app/routes/friends/new.js

```
import Ember from 'ember';

export default Ember.Route.extend({
  model: function() {
    return this.store.createRecord('friend');
  },
  activate: function() {
    console.log('----- activate hook called -----');
  },
  deactivate: function() {
    console.log('----- deactivate hook called -----');
  },
  // actions omitted for clarity
});
```

And then visit http://localhost:4200/friends/new[70] and click cancel or friends.

[66] http://emberjs.com/api/classes/Ember.Route.html
[67] http://emberjs.com/api/classes/Ember.Route.html#method_activate
[68] http://emberjs.com/api/classes/Ember.Route.html#method_deactivate
[69] http://emberjs.com/api/classes/Ember.Route.html#method_resetController
[70] http://localhost:4200/friends/new

We should see something like the following in our browser's console:

```
----- deactivate hook called -----
Rendering friends.index with default view <borrowers@view:default::ember721> Object {fullName: "view:friends.index"}
----- active hook called -----
Rendering friends.new with default view <borrowers@view:default::ember833> Object {fullName: "view:friends.new"}
>
```

Activate and Deactivate hooks

Coming back to our original problem of the unsaved record in the store, we can use the **deactivate** hook to clean up our code.

Let's rewrite **app/routes/friends/new.js** so the **deactivate** hook does what we expect:

Cleaning up the store on deactivate in app/routes/friends/new.js

```
import Ember from 'ember';

export default Ember.Route.extend({
  model: function() {
    return this.store.createRecord('friend');
  },
  deactivate: function() {
    // We grab the model loaded in this route
    //
    var model = this.modelFor('friends/new');

    // If we are leaving the Route we verify if the model is in
    // 'isNew' state, which means it wasn't saved to the backend.
    //
    if (model.get('isNew')) {

      // We call DS#destroyRecord() which removes it from the store
      //
      model.destroyRecord();
    }
  }
});
```

Another scenario where it is common to use the deactivate hook involves the **Edit Routes**. For example, if we try to edit a friend and don't save the changes but click cancel, the friend profile will still show whatever change we leave unsaved. To solve this problem we'll use the **deactivate** hook, but instead of checking if the model **isNew**, we'll call **model.rollback()**. This will return the attributes to their initial state if the model isDirty.

Using deactivate hook app/routes/friends/edit.js

```
import Ember from 'ember';

export default Ember.Route.extend({
  deactivate: function() {
    var model = this.modelFor('friends/edit');
    model.rollback();
  }
});
```

Tasks

We have the same problem on the **Articles Index Route**. Implement the **deactivate** hook so that any unsaved articles are not shown in the index.

Working with JavaScript plugins

In this chapter we'll learn how to write Ember helpers that can be consumed in our templates. To do so, we'll write a helper called **formatted-date** that will show the date when an article was borrowed. Instead of showing **Sun Sep 28 2014 04:58:30 GMT-0500**, we'll see **September 28, 2014**.

We'll implement **formatted-date** using Momentjs[71], a library that facilitates working with dates in JavaScript.

Installing moment

Remember that ember-cli uses Bower to manage frontend dependencies. Here we'll use the same pattern used to install **picnicss**: we'll add **moment** to Bower and then use **app.import** in our **Brocfile.js**.

We can also install front-end dependencies via npm if they are packed as addons. We'll learn more about this in a later chapter.

First, we install **moment**:

```
$ bower install moment --save
```

The option `--save` adds the dependency to our **bower.json**. We should find something similar to "**moment**": "∼**2.8.3**" (the version might be different).

Next, let's import **moment**. To find out which file to import, let's go to **bower_components/moment/**. We'll see that it contains a **moment.js** file that is the non-minified version of the library. We can also point to any of the versions under the directory **min/**. For now, let's use the non-minified.

Moment site also includes information when consuming via bower[72]

Let's add the following to our **Brocfile.js**:

[71] http://momentjs.com

[72] http://momentjs.com/docs/#/use-it/bower/.

```
app.import('bower_components/moment/moment.js');
```

Next, if we navigate to http://localhost:4200[73], open the console, and type "moment" we should have access to the **moment** object.

We have successfully included our first JavaScript plugin, but we need to be aware of some gotchas.

It's a global!

At the beginning of the book, we mentioned that one of the things ember-cli gives you is support to work with **ES6 Modules** rather than globals. It feels like taking a step backward if we add a library and then use it through its global, right?

The sad news is that not all libraries are written in such a way that they can be consumed easily via a modules loader. Even so, if there is an **AMD** definition included in the library, not all of them are compatible with the module loader used by **ember-cli**.

For example, **moment** includes an **AMD** version:

moment AMD definition

```
// ...
} else if (typeof define === 'function' && define.amd) {
    define('moment', function (require, exports, module) {
        if (module.config && module.config() && module.config().noGlobal === tru\
e) {
            // release the global variable
            globalScope.moment = oldGlobalMoment;
        }

        return moment;
    });
```

Unfortunately, the module loader ember-cli is using doesn't support that yet.

Other libraries do the following:

[73] http://localhost:4200

Anonymous module

```
define([], function() {
        return lib;
});
```

This is known as an anonymous module. Although its syntax is valid, the loader doesn't support this either because it expects named modules.

In the near future people will be able to use **moment** or other JavaScript libraries via **import**, but the integration is not yet ready yet. See issue #2177[74] for more info.

This issue is not entirely the fault of ember-cli, but in fact results from everyone building their libraries in different formats, making it difficult for consumers to use.

What can we do about it?

Wrapping globals

Instead of consuming globals directly, let's wrap them in a helper module that will allow us to foster the use of modules and to easily update or replace **moment** once we have a way to load it via the module loader.

First, let's create a utils file called **date-helpers**:

```
$ ember g util date-helpers
installing
  create app/utils/date-helpers.js
installing
  create tests/unit/utils/date-helpers-test.js
```

Replace **app/utils/date-helpers.js** with the following:

[74] https://github.com/stefanpenner/ember-cli/issues/2177

Wrapping globals: app/utils/date-helpers.js

```
function formatDate(date, format) {
  return window.moment(date).format(format);
}

export {
  formatDate
};
```

Here we are wrapping the call to **moment#format** in the function **formatDate**, which we can consume doing **import { formatDate } from 'utils/date-helpers'**;. With this, we are back to our idea of using modules. We'll also have the facility to easily update **moment** when our loader is ready to load it.

If we decide to stop using **moment** and replace it with any other similar library, we won't need to change our consuming code since it doesn't care how **format-date** is implemented.

Writing an Ember helper: formatted-date.

Helpers are pieces of code that help us augment our templates. In this case, we want to write a helper to create a date as a formatted string.

ember-cli includes a generator for helpers. Let's create **formatted-date** with the command **ember g helper formatted-date**, and then modify **app/helpers/formatted-date** so it consumes our format function.

Formatted Date helper

```
import Ember from 'ember';

// We are consuming the function defined in our utils/date-helpers.
import { formatDate } from '../utils/date-helpers';

export default Ember.Handlebars.makeBoundHelper(function(date, format) {
  return formatDate(date, format);
});
```

Once we have our helper defined, we can use it in the component **app/templates/components/articles/article-row.hbs**:

Using formatted-date in app/components/articles/article-row.hbs

```
<td>{{article.description}}</td>
<td>{{article.notes}}</td>
<td>{{formatted-date article.createdAt 'LL'}}</td>
<td>{{view "select" content=articleStates selection=article.state}}</td>
<td>
  {{#if article.isSaving}}
    <p>Saving ...</p>
  {{/if}}
</td>
```

Now, when we visit any of our friends' profiles, we should see the dates in a more attractive format.

Description	Notes	Borrowed since	
Mustache	this one Frank lent me since 1971	September 28, 2014	borrowed
My Guitar	wants to kill your mama	September 27, 2014	returned

Articles using formatted-date

Working with libraries with named AMD distributions.

Before the addons system existed, the easiest way to distribute JavaScript libraries to be consumed in ember-cli was to have a build with a named AMD version, importing the library using **app.import**, and whitelisting the library's exports.

Let's study ic-ajax[75], an "Ember-friendly **jQuery.ajax** wrapper." If we navigate to the lib/main.js[76], we'll notice that the source of the application is written with **ES6** syntax, but it is distributed[77] in different formats. This allows us to consume it in either global or module formats.

As mentioned previously, **loader.js** doesn't work with anonymous AMD distributions. If we want to include **ic-ajax**, we need to use the **named AMD** output. Let's try **ic-ajax** in our project for a first sketch of the dashboard.

First we need to remove **ember-cli-ic-ajax** from our **package.json** by running the following command:

[75] https://github.com/instructure/ic-ajax/tree/v2.0.1/lib
[76] https://github.com/instructure/ic-ajax/blob/master/lib/main.js
[77] https://github.com/instructure/ic-ajax/tree/v2.0.1/dist

Uninstalling a npm package

```
npm uninstall ember-cli-ic-ajax --save-dev
```

The library we just removed wraps all the steps we are about to perform, but we won't be using it. We are interested in learning how things work under the hood and what we gain when we use the addon.

Next we need to add the library to Bower. We can do so with `bower install ic-ajax --save`. Once it's installed, let's import it into our **Brocfile.js** as follows:

Importing ic-ajax

```
app.import('bower_components/ic-ajax/dist/named-amd/main.js');
```

ic-ajax's default export is the **request** function, which allows us to make petitions and manage them as if they were promises. Let's use this to create a "dashboard" object.

We'll present dashboard as the home page of our application, so when we navigate to the root url we'll see the reports. We already have the template, but let's create the route to load the required data. Create `app/routes/index.js` with the following content:

app/routes/index.js

```
import Ember from 'ember';
import request from 'ic-ajax';

export default Ember.Route.extend({
  model: function()  {
    return request('/api/friends').then(function(data){
      return {
        friendsCount: data.friends.length
      };
    });
  }
});
```

And then replace `app/templates/index.hbs` so it uses **friendsCount**:

```
<h1>Dashboard</h1>
<hr/>
<h2>Total Friends: {{model.friendsCount}}</h2>
```

The previous code is correct, but we'll see the following error when running **ember server**:

Error when importing ic-ajax

```
$ ember server --proxy http://api.ember-cli-101.com
version: 0.1.9
Proxying to http://api.ember-cli-101.com
Livereload server on port 35729
Serving on http://0.0.0.0:4200
ENOENT, no such file or directory '/borrowers/tmp/tree_merger-tmp_dest_dir-KIfHr\
FRc.tmp/ic-ajax.js'
Error: ENOENT, no such file or directory '/borrowers/tmp/tree_merger-tmp_dest_di\
r-KIfHrFRc.tmp/ic-ajax.js'
...
```

At the beginning of this chapter, we mentioned that part of the process of consuming named AMD libraries is to use **app.import** and **whitelist** the library's exports. We didn't explain what we meant by the latter.

During the build process, all our files under **app/** go through a transformation step where the ES6 modules are converted to AMD format. When something like **import request from 'ic-ajax'**; is found internally, the tool in charge of transpiling the code checks if that is something already registered in the module system. If not, it tries to find the module and convert it to the proper format. In the previous scenario, it will try to find a file called **ic-ajax.js**, but because it is a library we are including externally, such a file doesn't exist. This causes the build to fail.

Whitelisting in this context means telling the tool in charge of transforming our ES6 files to AMD that whenever **import request from 'ic-ajax'** is found, it is to assume its inclusion and refrain from resolving it.

To do so, we pass an option called exports to **app.import** that whitelists **ic-ajax** and its **exports**.

In the **Brocfile.js**, let's replace the call to **import** with the following:

Importing ic-ajax with exports

```
app.import('bower_components/ic-ajax/dist/named-amd/main.js', {
        exports: {
          'ic-ajax': [
            'default',
            'defineFixture',
            'lookupFixture',
            'raw',
            'request',
          ]
        }
      });
```

If we run **ember server**, we'll see that everything works. We can see the friends count in our dashboard by visiting http://localhost:4200/[78].

ember-cli-ic-ajax

We started this chapter by removing **ember-cli-ic-ajax**, an addon that wraps the call to import and include exports for us. If we inspect the index file in the addon[79], we'll notice that it has almost the same things we added to our **Brocfile.js**.

Now that we understand how importing named AMD libraries works, we can remove the **import** for **ic-ajax** from the **Brocfile.js** and use it via the addon. Let's run the following commands and then stop and start the server. Everything should work:

```
$ bower uninstall ic-ajax --save
$ npm i ember-cli-ic-ajax --save-dev
```

npm i is an alias for **npm install**

[78] http://localhost:4200/
[79] https://github.com/rwjblue/ember-cli-ic-ajax/blob/master/index.js#L18

A temporary replacement for moment.js

Let's consume a simple named AMD library that takes a date and returns its value after calling **.toDateString()**. This will be a simple example just to practice another module for importing named AMD.

The name of the library is **borrowers-dates** and it is located in https://github.com/abuiles/borrowers-dates[80].

The following is the content of the library:

borrowers-dates library

```
define("borrowers-dates", ["exports"], function(__exports__) {
  "use strict";
  function format(date) {
    return date.toDateString();
  }

  __exports__.format = format;
});
```

The library exports a function called **format**. Let's consume it via bower:

```
bower install borrowers-dates --save
```

And then import it through our **Brocfile.js**:

Consuming borrowers-dates

```
app.import('bower_components/borrowers-dates/index.js', {
  exports: {
    'borrowers-dates': [
      'format'
    ]
  }
});
```

With the library included, let's consume it in **app/utils/date-helpers.js** instead of moment:

[80] https://github.com/abuiles/borrowers-dates

Using borrowers-dates in app/utils/date-helpers.js

```
import { format } from 'borrowers-dates';


function formatDate(date) {
  return format(date);
}

export {
  formatDate
};
```

Now when we visit the profile for any our friends with articles, we'll see the dates rendered differently. This is because we are no longer using moment.

Tasks

Remove borrowers-dates and go back to using moment.

ember-browserify

Browserify[81] is a Node library which allows us to consume other Node libraries in the Browser using CommonJS (which is Node's module system), what this means is that we can install libraries like MomentJS using npm and then consume them in the browser via browserify. But wait, to use Browserify we actually need to install the library and create a "bundle" with our dependencies, normally we'll run something like `browserify main.js -o bundle.js` and then use `bundle.js` via a script tag `<script src="bundle.js"></script>`.

As we can imagine this can get tricky and hard to manage in our ember-cli application, but thanks to Edward Faulkner[82] there is addon which allow us to consume libraries from npm with browserify without needing us to worry about the bundling process, it is called ember-browserify[83].

Using ember-browserify

First we need to install the addon, which we can do running `ember install`:

[81] http://browserify.org/
[82] https://github.com/ef4
[83] https://github.com/ef4/ember-browserify

```
$ ember install:addon ember-browserify
```

Once the addon has been installed, we are going to use it to consume MomentJS from npm in our `date-helpers` file.

Before doing that, let's make sure we have removed `moment` from our `bower.json` and also that we have removed `app.import('bower_components/moment/moment.js');` from the `Brocfile`.

Next, let's install moment via npm, which we can do with `npm install moment --save-dev`.

Once it has been installed we can consume it from npm thanks to ember-browserify just doing `import moment from 'npm:moment';`.

Let's use it in our `date-helpers` so `formatDate` uses `moment`.

app/utils/date-helpers.js

```
import moment from 'npm:moment';

function formatDate(date, format) {
  return moment(date).format(format);
}

export {
  formatDate
};
```

And that's it, we are now consuming MomentJS via browserify just as if it was other module in our application.

Wrapping up

In this chapter we have covered how to work with JavaScript plugins both as globals, consuming named AMD plugins and via ember-browserify.

We didn't cover how to write reusable plugins to be consumed with ember-cli. This is what addons are used for, and we'll talk about them in the next chapter.

The API for consuming third-party plugins is not 100% finished in ember-cli, and this chapter might change along with its development. The story is still a work in progress, but the goal is to make it easier to work with any plugin regardless of the format in which it was written.

Components and Addons

Web Components

Web Components are a new mechanism that allows us to extend the DOM with our own elements rather than limit ourselves to traditional tags. We can define our own tags, wrapping up all the display logic in a single bundle, and reuse it between different applications. We'll use the component as any other tag and the browser will understand how to render it based on its definition.

Let's examine how a Share on Twitter button works. Currently, we need to include some JavaScript and then create an anchor tag that will be transformed by the JavaScript snippet:

Twitter Share Button

```
<a class="twitter-share-button"
  href="https://twitter.com/share">
Tweet
</a>
<script type="text/javascript">
window.twttr=(function(d,s,id){var t,js,fjs=d.getElementsByTagName(s)[0];if(d.ge\
tElementById(id)){return}js=d.createElement(s);js.id=id;js.src="https://platform\
.twitter.com/widgets.js";fjs.parentNode.insertBefore(js,fjs);return window.twttr\
||(t={_e:[],ready:function(f){t._e.push(f)}})}(document,"script","twitter-wjs"))\
;
</script>
```

The previous code will be the same regardless of the application we are developing. Sounds like a good candidate for a component since it's a chunk of code that can be reused across applications. A possible twitter-button component could look something like the following:

Twitter Share Button

```
<twitter-button>
  Ember.js Rocks!
</twitter-button>
```

All the implementation details are hidden in its definition. As consumers, we are only interested in the final product and we needn't worry about how it is accomplished.

Web Components are a great tool that we can use to write more expressive applications and to avoid code repetition within projects. Unfortunately, it is not yet supported in all browsers.

To tackle this problem, Ember introduced the concept of Components. This is an API that allows us to write components today by following the **W3C** specification as closely as possible. The day components become widely available, we'll be able to switch without hiccups.

The official name for Web Components in the **W3C** is (Custom Elements)[http://w3c.github.io/webcomponents/spec/custom/#about].

ember-cli addons

ember-cli has a built-in mechanism that allows us to augment **ember-cli**'s functionality and share code easily between different applications. This mechanism is known as addons.

Using addons, we can easily write Ember Components and share them with others using npm. Let's create our first component, which will help us grab an image to use as placeholder in our friends' profiles.

ember-cli-fill-murray

http://www.fillmurray.com is a service we can use to get random images of Bill Murray to use as placeholders. Let's write an addon so that we can do something like the following in any of our templates:

Fill Murray Component

```
{{fill-murray width=300 height=300}}
```

First we need to create the addon. **ember-cli** has a command for this. Outside of our borrowers directory, let's run the following:

Creating an ember-cli addon

```
$ ember addon ember-cli-fill-murray
version: 0.1.9
installing..
  create .bowerrc
  create .editorconfig
  create .ember-cli
  create tests/dummy/
  ...
```

The **addon** command creates a directory very similar to the one created by **new**, but the former is done in a way that allows it to be distributed as an addon.

If we go to the directory **ember-cli-fill-murray**, it will look like the following:

Addon directory

```
.
|-- Brocfile.js
|-- LICENSE.md
|-- README.md
|-- addon
|-- app
|-- bower.json
|-- bower_components
|-- config
|-- index.js
|-- node_modules
|-- package.json
|-- testem.json
|-- tests
+-- vendor
```

If we open **package.json**, we'll see the following section:

```
  "keywords": [
    "ember-addon"
  ],
```

That's how **ember-cli** detects the presence of an **addon**. When we include the library in an **ember-cli** project, it will transverse the dependencies and identify as **addons** the items with the keyword **ember-addon**. We'll also use **package.json** to specify any dependency our library might have.

Next we have **index.js**, which is the entry point for loading our **addon**. If we need to add any extra configuration for our addon, we'll specify it in this file. For now let's work with the basic one, which looks like this:

```
module.exports = {
  name: 'ember-cli-fill-murray'
};
```

Next we have the directories **app** and **addon**. This is where the code for our **addon** will live.

Whatever we put into **app** will be merged into our application's namespace, meaning we'll consume it just as if it were inside our **ember-cli** project.

For example, if we had an **app/model/friend-base.js** file in our **addon**, we could consume it in any of the models in our **borrowers** app thusly:

Consuming an addon's app/model/friend-base.js

```
import FriendBase from './friend-base';

...
```

If we had put **friend-base.js** into the **addon** directory, instead of getting merged into the consuming application namespace, it would be kept under the **addon namespace** with the previous example. If our **addon** was called **borrowers-base** and we had **addon/models/friend-base.js**, then we would have consumed it like this:

Consuming modules from addon's namespace

```
import FriendBase from 'borrowers-base/models/friend-base';

...
```

Going back to our **ember-cli-fill-murray addon**, we have the directory in place and we want to distribute a component called **fill-murray**. Inside the directory, we can also use **ember-cli** generators. We'll do that in order to create the component:

Bill Murray Component

```
$ ember generate component fill-murray
version: 0.1.5
installing
  create app/components/fill-murray.js
  create app/templates/components/fill-murray.hbs
installing
  create tests/unit/components/fill-murray-test.js
```

In **app/components/fill-murray.js**, we can specify the properties for our component:

app/components/fill-murray.js

```
import Ember from 'ember';

export default Ember.Component.extend({
  height: 100, // Default height and width 100
  width: 100,

  //
  // The following computed property will give us the url for
  // fill-murray. In this case it depends on the properties height and width.
  //

  src: Ember.computed('height', 'width', function() {
    var base = 'http://www.fillmurray.com/';
    return base + this.get('width') +  '/' + this.get('height');
  })
});
```

Next we need to specify the body of our component in **app/templates/components/fill-murray.hbs**:

```
<img src={{src}}>
```

When rendering the component, it inserts an **img** tag that reads the source from the computed property we specified.

Our **addon** is now ready. The next step is to distribute it via npm. First let's change the name in **package.json** because **ember-cli-fill-murray** is already taken. We'll use **ember-cli-fill-murray-your-github-nickname** and set version to **0.1.0**. It will look something like this:

```
  "name": "ember-cli-fill-murray-your-github-nickname",
  "version": "0.1.0",
```

With the previous values in place, let's do **npm publish**. Our **addon** is now ready to be consumed.

Consuming fill-murray in borrowers

Once our package is in **npm**, we can add it to our application by running the following command:

```
$ npm install --save ember-cli-fill-murray-your-github-nickname
```

Once it is installed, we can consume the component in any of our templates as follows:

```
{{fill-murray width=150 height=150}}
```

Let's use it in our friend template. First we want to add some styling for our friend template in **app/styles/app.css**:

```
.friend-profile{
  text-align: left;
}

.friend-profile img {
  float:left;
  margin:1em 2em 1em 1em;
}

.friend-profile-links li {
  display: inline;
  list-style-type: none;
  padding-right: 20px;
}
```

Then modify **app/templates/friends/show.hbs** to look like the following:

Consuming fill-murray in app/templates/friends/show.hbs

```
<div class="friend-profile" class="row">
  <div class="friend-info full">
    {{fill-murray width=300 height=300}}
    <div>
      <p>{{model.fullName}}</p>
      <p>{{model.email}}</p>
      <p>{{model.twitter}}</p>
      <ul class="friend-profile-links">
        <li>{{link-to 'Edit info' 'friends.edit' model}}</li>
        <li>{{link-to 'Lend article' 'articles.new'}}</li>
        <li><a href="#" {{action "delete" model}}>Delete</a></li>
      </ul>
    </div>
  </div>
</div>

<div class="articles-container">
  {{outlet}}
</div>
```

After editing the template, the Bill Murray placeholder will appear when we visit a friend's profile.

Ember Data

In this chapter we'll cover some of the public methods from the DS.Store[84] and learn how to load relationships asynchronously.

DS.Store Public API

The store is the main interface we'll use to interact with our records as well as the backend. When we create, load, or delete a record, it is managed and saved in the store. The store then takes care of replicating any change to the backend.

We won't cover all of the functions, but we'll go over the more common ones and their gotchas.

all

store.all is similar to **store.findAll**, but instead of making a request to the backend it returns all the records already loaded in the store. The result of this method is a **live array**, which means it will update its content if more records are loaded into the store for the given type.

Let's study this with the inspector by navigating to http://localhost:4200 and clicking refresh. Next we'll grab an instance of the application route and run the following commands in the console:

```
friends = $E.store.all('friend')
friends.get('length')
> 0
friends.mapBy('firstName')
> []
```

We stored the result in a variable called friends, and we'll notice that the length of the collection is zero. This makes sense because we haven't loaded any **friends** yet. Click on the friends link and run the following:

```
friends.get('length')
> 3
friends.mapBy('firstName')
> ["zombo Wamba", "Pizza", "Loading-this"]
```

We'll see that the result is no longer zero. When we navigated to the friends route, a request to the backend was made and some records were loaded into the store. As we mentioned, the result from **all** is a **live array**. That's why our **friends** variable was updated without requiring any additional steps.

[84] http://emberjs.com/api/data/classes/DS.Store.html

filter

The **filter** function behaves similarly to **findAll** however, in addition to the type, it also takes a parameter known as the **filter function**. The filter function is called once for every record in the result and returns those for which the filter function returns true.

By default, filter will work against elements already loaded into the store (like **all**). If we want to force a request to the backend, we can pass an object and it will make a request with every property on the object as a parameter.

Again, let's see this working on our application by navigating to http://localhost:4200/friends and putting the following in the console:

```
friends = $E.store.filter('friend', function(friend){
   return friend.get('totalArticles') % 2 == 0
})
friends.mapBy('firstName')
```

The previous call to filter will take every friend already loaded in the store and then select the ones who have borrowed an even number of articles.

Let's suppose our API supports a parameter **hasArticles**, which return only the friends who currently have an article. Let's say we want to filter them by the ones who have an even number of articles.

We could write the filter as follows:

```
friends = $E.store.filter('friend', {hasArticles: true}, function(friend){
   return friend.get('totalArticles') % 2 == 0
})
> GET "http://localhost:4200/api/v2/friends?hasArticles=true".
```

If we inspect our network tab, we'll see the following **GET** request to the server:

```
http://localhost:4200/api/v2/friends?hasArticles=true
```

Once the records are loaded, it will apply the filter and return the values returning true for the given filter function.

The result from a filter function is a **live array** as well. In our example, if any of our friends borrow a new article and the total number of articles is odd, then it will disappear from our result.

XHR logging in the console is a great way to debug our applications. We can enable it using the setting in Chrome's DevTools. See slide #4 in the presentation [Wait, DevTools could do THAT? by Ilya Grigorik](https://www.igvita.com/slides/2012/devtools-tips-and-tricks/#4](https://www.igvita.com/slides/2012/devtools-tips-and-tricks/#4).

findAll

If we call **findAll** with a model name, then it will make a request to load a list of records of that type. The following is an example:

```
friends =  $E.store.findAll('friend')

XHR finished loading: GET "http://localhost:4200/api/v2/friends".
```

If we want to send query parameters with the request, then we should use `store.findQuery`, it receives the name of the model and an object as second argument, every key on the object will be included as parameter:

```
friends =  $E.store.findQuery('friend', {hasArticles: true, sort_by: 'created_at\
'})

XHR finished loading: GET "http://localhost:4200/api/v2/friends?hasArticles=true\
&sort_by=created_at".
```

In the previous request we asked **findQuery** to load all the articles, sending as parameters the keys **hasArticles** and **sort_by**.

Like **all** and **filter**, the result from **findAll** is a **live array**. When called, it makes a request to the server and the collection is updated when more records are added to or removed from the store.

find: Loading a single record

If we want to load a single record then we should use **store.find** . To do that, we use the name of the model and the record's **id** as second argument:

```
$E.store.find('friend', 15)
XHR finished loading: GET "http://localhost:4200/api/v2/friends/15".
```

In the previous example, we loaded the friend with id 15. The **store** will only make a request to the server if the friend is not available in the store. To understand this, let's go to http://localhost:4200/friends and try the following in the console:

```
id = $E.store.all('friend').get('firstObject').id
$E.store.find('friend', id)
```

If we open our network tab, we'll see that the store didn't make any requests this time. This is because we asked for a friend who was already loaded into the store.

It's important to mention that **find**, **all**, and **filter** return promises. When testing on the browser's console, we don't have to worry about it, but if we want to use the result in our application then we need to keep this in mind.

getById

We can use **store.getById('friend', 15)** to fetch a user directly from the store. Unlike find, findQuery or findAll, the behavior of this function is synchronous. It will return the record if it is available, or null otherwise.

metadataFor

If our API includes a "meta" key with a response, we can access such metadata with the **metadataFor** function. This is useful when we implement things like pagination.

Suppose the response from our API is something like the following when we fetch all of our friends:

```
{
  friends: [ ... ],
  meta: { total: 30}
}
```

We can then read the meta key doing **this.store.metadataFor('friend')**;

createRecord

We are already familiar with **createRecord**, which is used when we want to create a new record for a given type. For example:

```
this.store.createRecord('friend', {attrs..});
```

We can also use createRecord via a relationship. Suppose we are in the context of a friend and we know they have an **articles** property that represents all the articles belonging to another friend. If we want to add a new article, we can do it using the following syntax:

```
friend.get('articles').createRecord({attrs...});
```

This won't work if the relationship is **async**.

Loading relationships

We already covered how to specify relationships between models. If we are defining a relationship of type "has many", then we use the keyword **hasMany**. If we want a "belongs to", we use **DS.belongsTo**.

We switched to v2 of the API, which side-loads all the article records for our friend but didn't stop to understand how that worked.

There are two ways to work with relationships in Ember. The first is working with records pre-loaded into the store, and the second is to load them on demand.

With the first strategy, we'll specify the ids on the payload of the records that the model is related to. Ember-Data looks for those records and fills up the association automatically. Under this model, the records that are part of the association need to be loaded into the **Store** or **side-loaded** with the parent.

If we inspect the payload for friends arriving in version 2 of the API, the results look something like the following:

```
{
  id: 48,
  first_name: "zombo",
  last_name: "Pombo",
  email: "zombo@pombo.com",
  twitter: "zombo",
  total_articles: 2,
  article_ids: [
    40,
    41
  ]
}
```

This includes the model's attributes and a key called **article_ids**. This is what Ember-Data uses to bind the models.

Ember-Data expects the **articles** with ids 40 and 41 to be in the **Store**. If we do **store.getById('article', 40)**, it returns a value or else expects to have a key in the response called **articles** that includes the **articles** with id 40 and 41.

If the records are not present, we'll get the following error:

```
route: articles.index Assertion Failed: You looked up the 'articles'
relationship on a 'friend' with id 48 but some of the associated
records were not loaded. Either make sure they are all loaded together
with the parent record, or specify that the relationship is async
(`DS.hasMany({ async: true })`)
```

The **payload** when side-loading a relationship looks like the following:

```
{
  friend: {
    id: 48,
    first_name: "Wamba",
    last_name: "Pombo",
    email: "zombo@pombo.com",
    twitter: "zombo",
    total_articles: 2,
    article_ids: [
      41
    ]
  },
  articles: [
    {
      id: 41,
      created_at: "2014-10-17T16:04:51.884Z",
      description: "Pombo Set",
      state: "borrowed",
      notes: null,
      friend_id: 48
    }
  ]
}
```

The response above brings the **friend** record with id **48** and includes all of their articles.

This strategy for loading records works well if we know that all the records the association depends on are already in the store, or if there is a low number of records to side-load.

What if we want to load thousands of relationships in addition to implementing strategies like pagination or search? Enter **async** relationships.

In the previous error thrown by ember-data, the following was included: **specify that the relationship is async (DS.hasMany({ async: true }).**

Working with async relationships in Ember-Data

Ember-Data offers support for working with asynchronous relationships. All we have to do is mark the attribute as async. Then we can include the ids or an URL from which to load the records.

First it loads the parent record. Then it will load the records in the relationship, but only when we explicitly call the attribute. For example, if we call **friend.get('articles')**, Ember-Data will check if the **articles** are already loaded. If they are not, it will make a **GET** request. If the ids in the relationships are 40 and 41, then the **GET** request is going to be something like **/api/articles?ids%5B%5D=40&ids%5B%5D=41**.

Let's try this on our applications. First we'll update our **application adapter** to use **v3** of the API. Let's change **app/adapters/application.js**:

Using borrowers API V3

```
import DS from 'ember-data';

export default DS.ActiveModelAdapter.extend({
  namespace: 'api/v3'
});
```

If we check the response from http://api.ember-cli-101.com/api/v3/friends.json[85], we'll notice that this time the articles are not being side-loaded.

Next we need to update our friend model. We'll add the object **{async: true}** as second argument to the **hasMany** attribute for articles:

Specifying articles as async: app/models/friend.js

```
import DS from 'ember-data';
import Ember from 'ember';

export default DS.Model.extend({
  articles:      DS.hasMany('articles', {async: true}),
  email:         DS.attr('string'),
  firstName:     DS.attr('string'),
  lastName:      DS.attr('string'),
  totalArticles: DS.attr('number'),
  twitter:       DS.attr('string'),
  fullName: Ember.computed('firstName', 'lastName', function() {
    return this.get('firstName') + ' ' + this.get('lastName');
  })
});
```

[85] http://api.ember-cli-101.com/api/v3/friends.json

We just switched our model from working with side-load relationships to **async**.

Let's explore how **async** relations behave. If we navigate to http://localhost:4200/friends, click on any of our friends, and open the console, we'll see something like the following:

```
XHR finished loading: GET "http://localhost:4200/api/v3/articles/34".
XHR finished loading: GET "http://localhost:4200/api/v3/articles/35".
XHR finished loading: GET "http://localhost:4200/api/v3/articles/36".
XHR finished loading: GET "http://localhost:4200/api/v3/articles/16".
```

This time we didn't get the error because our articles were not loaded. Instead, Ember-Data made a **GET** request for each of our friends.

Will Ember-Data make 10,000 requests to our API if our friend has 10,000 items? No. We can tell Ember-Data to coalesce all those calls into a single request by setting the adapter's property **coalesceFindRequests** to true. Let's change **app/adapters/application.js** to the following:

Enable coalesceFindRequests

```
import DS from 'ember-data';

export default DS.ActiveModelAdapter.extend({
  namespace: 'api/v3',
  coalesceFindRequests: true
});
```

If we refresh the route, this time we'll see the following GET request:

```
XHR finished loading: GET
"http://localhost:4200/api/v3/articles?ids%5B%5D=34&ids%5B%5D=35&ids%5B%5D=36&id\
s%5B%5D=16".
```

Now it makes a single request to the API by passing the query parameter **ids** with all the articles that it needs to load.

In the following commit, we can check the backend implementation to load a list of articles if the ids parameter is present: abuiles/borrowers-backend- Add version 3[86].

[86] https://github.com/abuiles/borrowers-backend/commit/857cb40e654b8243b6e842a2bc78408cd50a9f4d#diff-b4f73470ac000871615a9c310e2537fcR5

Using links instead of ids.

There is another way to load relationships asynchronously in Ember-Data without specifying the ids. We can return a property called **links** with an object including an URL for each of the relationships to load asynchronously. Ember-Data will then make a request to the URL when we ask for the relationship records.

We'll move to version 4 of our API, which specifies the relationships using links.

To try this in our application, we'll update **application adapter** to use **v4** of the API. Let's change **app/adapters/application.js**:

Using borrowers API V4

```
import DS from 'ember-data';

export default DS.ActiveModelAdapter.extend({
  namespace: 'api/v4',
  coalesceFindRequests: true
});
```

If we look at the payload for v4 http://api.ember-cli-101.com/api/v4/friends.json, we'll noticed that it looks like the following:

JSON payload with links

```
{
  id: 48,
  first_name: "Zombo",
  last_name: "Pombo",
  email: "zombo@pombo.com",
  twitter: "zombo",
  total_articles: 2,
  links: {
    articles: "/api/v4/articles?friend_id=48"
  }
}
```

This time we don't have ids but an URL from which Ember-Data can load the relationship. If we go to a friend's profile, we'll see the request **GET "http://localhost:4200/api/v4/articles?friend_id=48"** in the network tab.

One important item to mention is that the request to load **async** data will only happen once. If we visit a friend's profile, go back to the friends index, and then visit that friend profile again, we

won't see a request to fetch the articles because Ember-Data will identify such a request as already fulfilled.

If we always want to load the records from the model hook on the **Articles Index Route**, then we can put a guard, check if the request is fulfilled, and if that's the case then force a reload. We can use something like the following:

app/routes/articles/index.js

```
import Ember from 'ember';

export default Ember.Route.extend({
  model: function() {
    var articles = this.modelFor('friends/show').get('articles');

    //
    // The return value from an async relationship is a PromiseArray.
    // The property isFulfilled will become true when the proxied
    // promise has been fulfilled. In this case, that would be when we
    // get a response from the API.
    //

    if (articles.get('isFulfilled')) {
      articles.reload();
    }

    return articles;
  },
  actions: {
    save: function(model) {
      model.save();
      return false;
    }
  }
});
```

If we try again, we'll see that navigating to a friend's profile will always cause a request to the API to fetch the articles.

The property **isFulfilled** is part of a set of properties included in the **PromiseArray** via the (Ember.PromiseProxyMixin[87].

Ember.PromiseProxyMixin has the following properties that we can use to guide the flow of our application:

[87] http://emberjs.com/api/classes/Ember.PromiseProxyMixin.html#property_isFulfilled

isFulfilled
isPending
isRejected
isSettled

What to use?

So many options. What should we use? It depends on our scenarios and how we want to load our data. Side-loading works perfectly when we are not fetching many records, but it can make your API really slow if you are returning a lot of relationships and a lot of records.

Async helps us alleviate the issue when we have a lot of records. This can help us keep our end-points lighter, but it might add some overhead when getting all the ids in a relationship.

The faster option from an API point of view would be to use links. This won't require the parent to know anything about its children, but then we lose other benefits.

For example, when using ids, Ember-Data will only load records from the server that are not yet available in the store. However, if some of the records are loaded, it won't make that request. With links, you lose that benefit because Ember-Data doesn't have any information. It will make the request and load data that you might already have available.

Again, it's a matter of weighing risks and benefits and finding what works best for us. We need to measure and experiment with different strategies before choosing the one that gives us the best performance.

Computed Properties and Observers

We already covered computed properties, which we use in different parts of our applications. One of these uses occurs on the friend model:

app/models/friend.js

```
import DS from 'ember-data';
import Ember from 'ember';

export default DS.Model.extend({
  // ...
  fullName: Ember.computed('firstName', 'lastName', function() {
    return this.get('firstName') + ' ' + this.get('lastName');
  })
});
```

With the code above, we created a new property on the model called **fullName** that depends on **firstName** and **lastName**. The computed properties are called once at the beginning and the result is cached until any of the dependent properties change.

Next we'll talk about a couple of features and things to keep in mind when defining computed properties.

Computed Property function signature

The functions we've used to declare a computed property have looked like the following:

Computed Property Function

```
fullName: Ember.computed('firstName', 'lastName', Ember.function() {
  return this.get('firstName') + ' ' + this.get('lastName');
})
```

Using the previous signature in the **function** we passed to `Ember.computed` we get computed properties working, but we can optionally specify it like so:

Computed Property Function

```
fullName: Ember.computed('firstName', 'lastName', function(key, value, oldValue)\
 {
  return this.get('firstName') + ' ' + this.get('lastName');
}
```

Now we can add support for setting the value of a computed property and handling how it should behave. The following is an excerpt from the Ember documentation where **firstName** and **lastName** are used:

Computed Property with set support

```
fullName: Ember.computed('firstName', 'lastName', function(key, value, oldValue)\
 {
  if (arguments.length === 1) {
    //
    // Works as getter
    //

    return this.get('firstName') + ' ' + this.get('lastName')
  } else {

    //
    // Works as setter
    //

    var name = value.split(' ');

    this.set('firstName', name[0]);
    this.set('lastName', name[1]);

    return value;
  }
})
```

For the curious, the following class has the implementation for computed property[88].

Why didn't we mention that we can use a computed property as setter? This is a very uncommon scenario that tends to cause a lot of confusion for people. Ideally, we use computed properties as

[88] https://github.com/emberjs/ember.js/blob/v1.7.0/packages/ember-metal/lib/computed.js#L78

Read-Only. In a later version of Ember, this might be the default. Stefan Penner[89] created an issue that aims to make computed properties Read-Only by default: default readOnly CP #9290[90].

Computed Properties gotchas

Computed properties and observers are normally fired whenever we call `this.set()` on the property they depend on. The downside of this is that they will be recalculated even if the value is the same.

Fortunately for us, Gavin Joyce[91] wrote an **ember-cli-addon** called ember-computed-change-gate[92] that offers an alternative function to define computed properties and that fixes observers such that they are only called if the property they depend on has changed.

We can install the addon with `npm i ember-computed-change-gate --save-dev` and use it in our friends model like so:

Using ember-computed-change-gate in app/models/friend.js

```
import DS from 'ember-data';
import Ember from 'ember';
import changeGate from 'ember-computed-change-gate/change-gate';

export default DS.Model.extend({
  //
  // Currently changeGate only support one property
  //
  capitalizedFirstName: changeGate('firstName', function(firstName) {
    return Ember.String.capitalize(firstName);
  })
});
```

Now our computed property `capitalizedFirstName` will be called only when the value of the dependent key has changed to a different value.

Observers

Ember has a built-in implementation of the Observer pattern[93], which allows us to keep track of changes in any property or computed property.

We use observers to implement auto saving in the article-row component with the following:

[89] https://twitter.com/stefanpenner
[90] https://github.com/emberjs/ember.js/issues/9290
[91] https://twitter.com/gavinjoyce
[92] https://github.com/GavinJoyce/ember-computed-change-gate
[93] http://en.wikipedia.org/wiki/Observer_pattern

app/components/articles/article-row.js

```
stateChanged: Ember.observer('model.state', function() {

  if (this.get('model.isDirty') && !this.get('model.isSaving')) {
    Ember.run.once(this, this.autoSave);
  }
}).on('init')
```

We define an observer calling `Ember.observer` which receives any number of properties to observe and the function to call when any of the properties change.

Calling `.on('init')` at the end of the observer definition is a common method to make sure the observer is enabled. By default, observers are not switched on until the function where they are defined is called. If we define the observer as follows:

```
stateChanged: Ember.observer('model.state, function() {
  if (this.get('model.isDirty') && !this.get('model.isSaving')) {
    Ember.run.once(this, this.autoSave);
  }
})
```

Then the observer won't have any effect until the function `stateChanged` is called. To make sure the observer is enabled we use `on('init')`, which calls the function as soon as the object where the function is defined gets created. In our example, that would be when an instance of `"app/components/articles/article-row.js` is created.

We might find some examples where observers are set calling `.observer('property')` at the end of a function definition. This pattern is valid but it relies on a mechanism called prototype extensions which might get removed in future versions of Ember. Please refer to the following pull request for more information emberjs/guides/pull/110[94]

We can also create an observer using `addObserver` from Ember.Observable[95]. We could define the `stateChanged` observer like this:

[94] https://github.com/emberjs/guides/pull/110
[95] http://emberjs.com/api/classes/Ember.Observable.html

```
  setObserver: function() {
    this.addObserver('model.state', this, this.stateChanged);
    // we need to call get on the property so the observers are setup
    // see for more info http://cl.ly/1f0Y1v2A1G04

    this.get('model').get('state');
  }.on('init'),
  stateChanged: function() {
    if (this.get('model.isDirty') && !this.get('model.isSaving')) {
      Ember.run.once(this, this.autoSave);
    }
  }
```

Observing collections

Ember adds two convenient properties to collections. We can use them if we want to observe changes to any of the members' properties, or if we want to do something every time an element is added or removed.

The first property is .[][96], which is just a special handler that changes every time the collection content changes.

The second one is @each[97], which allows us to observe properties on each of the items in the collection.

We can use the previous function in our articles index to call a function when we add a new article, and then other one when we change the state of an article:

app/controllers/articles/index.js

```
import Ember from 'ember';

export default Ember.Controller.extend({
  contentDidChange: Ember.observer('model.[]', function() {
    console.log('Called when we add or removed an article.');
  }),
  stateDidChange: Ember.observer('model.@each.state', function() {
    console.log('Called when the state property change for any of the articles.'\
);
  })
});
```

[96] http://emberjs.com/api/classes/Ember.Array.html#property__

[97] http://emberjs.com/api/classes/Ember.Array.html#property__each

If we visit any of our friends' profiles and change the state for any article or add a new one, we'll see the relevant messages in the browser's console.

Driving our application state through the URL

In JSConf EU 2013, Tom Dale[98] gave a talk called Stop Breaking the Web[99].

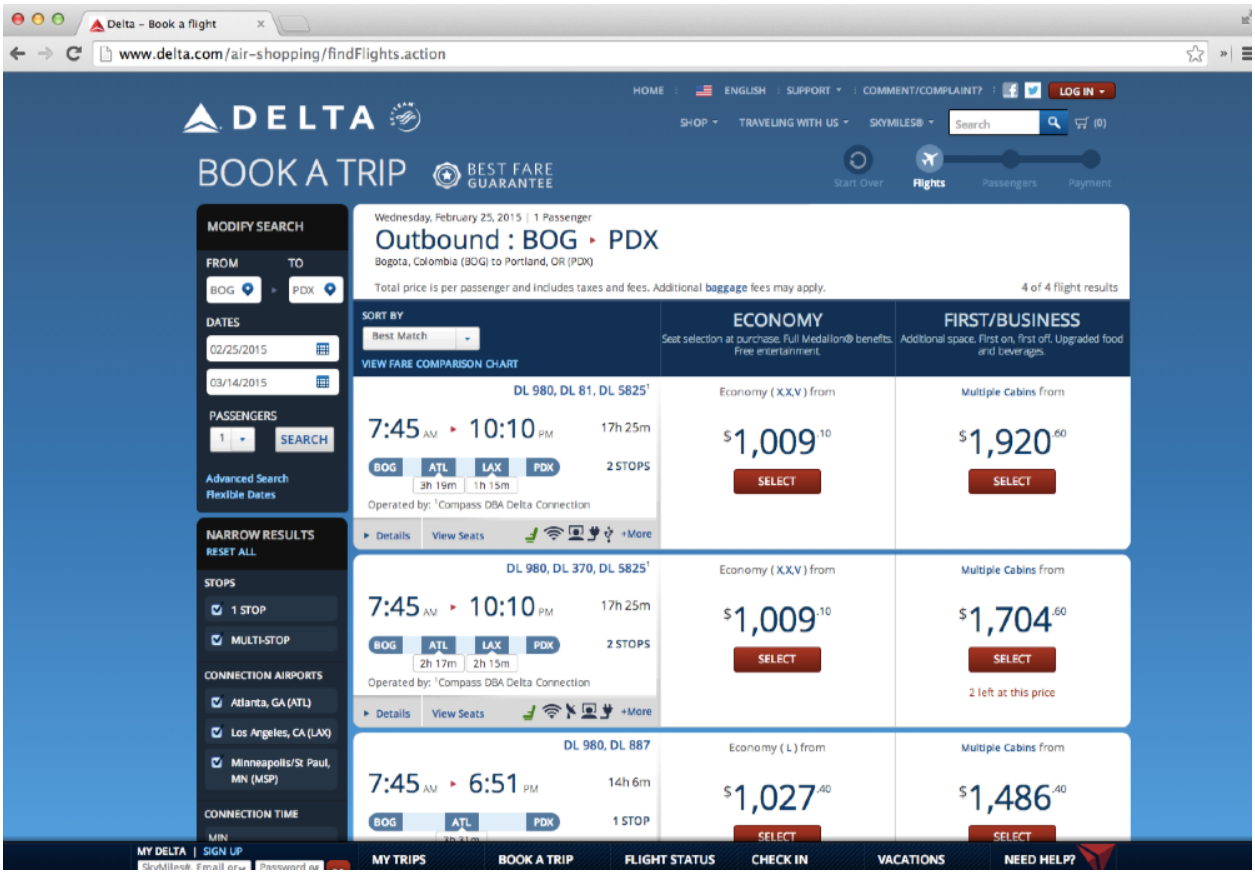
Tom talks about the importance of the URL and how we should give it a higher priority in our applications. Ideally, the URL should be able to reflect our application state in such a way that we can easily reference it, bookmark it, or share it with others.

Some of us have probably experienced some frustration when visiting a website that has search functionality but loses our selections between page reloads, or that doesn't allow us to easily share what we see with others.

Airline websites offer an example of this issue. The following image shows Delta's website after searching for flights to the next EmberConf.

[98] https://twitter.com/tomdale

[99] http://2013.jsconf.eu/speakers/tom-dale-stop-breaking-the-web.html

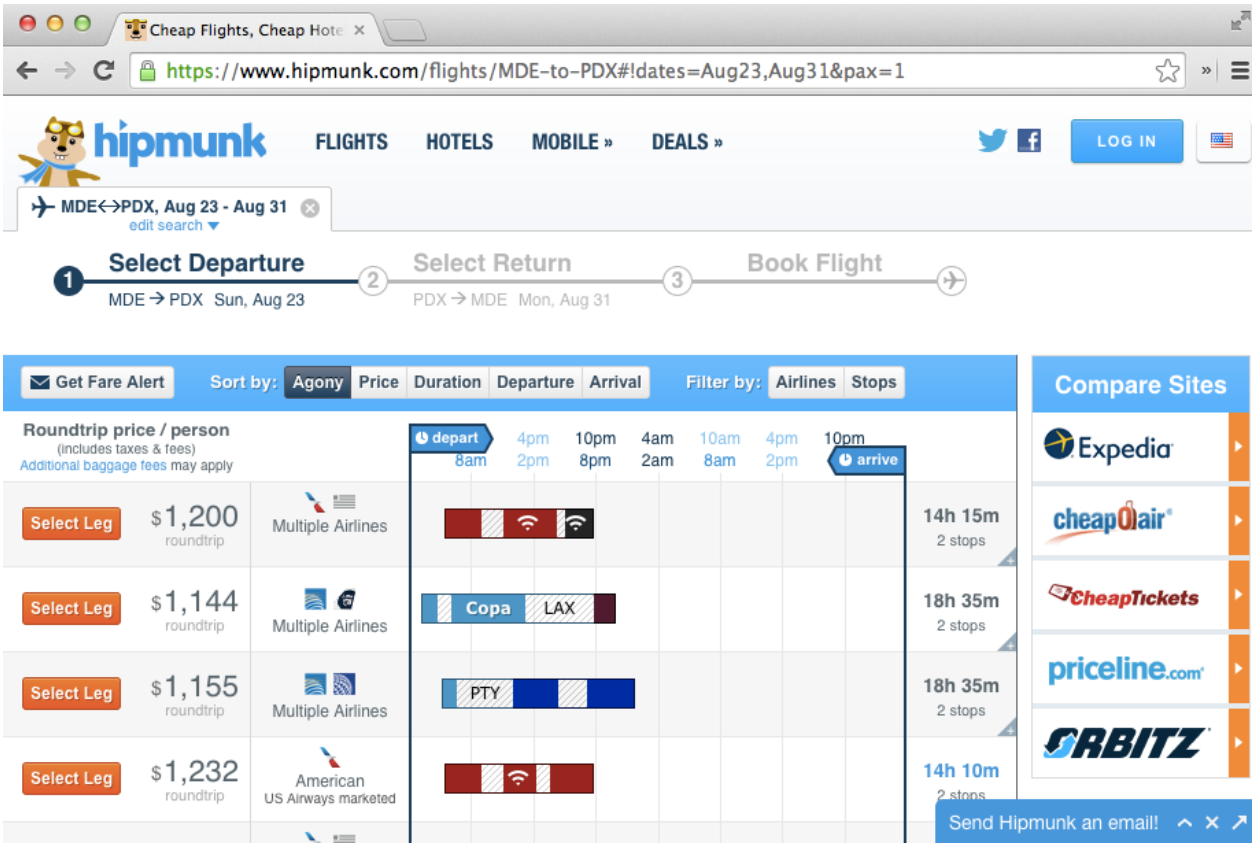


Search on Delta

The URL after the search is http://www.delta.com/air-shopping/findFlights.action, which doesn't really tell us anything about the screen we are visiting. If we copy and paste the URL in other browser, we'll get a bunch of errors and not the search we originally performed.

Now let's do a search on hipmunk[100]. This website places greater value on the functionality of the URL.

[100] https://www.hipmunk.com



hipmunk search

The search above results in the following: flights/MDE-to-PDX#!dates=Aug23,Aug31&pax=1[101]. Isn't that beautiful? Just by reading the URL, we know our destination and the dates of our trip. Clicking the URL takes us to the original search we see in the image. Suppose we want someone to buy the ticket for us; we can simply share the URL and be done with it.

Ember also appreciates the beauty of a functional URL. In fact, our applications are driven by URLs that we specify in **app/router.js**. This doesn't mean we are immune from building bad applications that don't respect the URL, but at least it gives us the tools to avoid these issues and invites us to think better about our URLs.

Sorting friends.

When visiting the friends index, we want to be able to sort them by clicking on the **Name** or **Articles** column and then toggle ascending or descending between clicks.

We'll add 2 properties to our friends index controller: **sortBy** and **sortAscending**.

To change our sort field dynamically, we will create an action `setSortBy` that will receive as parameter the field we want to sort our properties by.

[101] https://www.hipmunk.com/flights/MDE-to-PDX#!dates=Aug23,Aug31&pax=1

We'll also toggle the property `sortAscending` every time we call the action `setSortBy`. For example, if it's `true` then it becomes `false` and vice versa.

app/controllers/friends/index.js

```
import Ember from 'ember';

export default Ember.Controller.extend({
  sortAscending: true,
  //
  // We'll use sortBy to hold the name of the field we want to sort by.
  //
  sortBy: 'firstName',
  actions: {
    //
    // The setSortBy function receives the name of the function and
    // toggle `sortAscending`. The function `toggleProperty`  comes from the
    // [Observable Mixin](http://emberjs.com/api/classes/Ember.Observable.html)
    // it switches a boolean property between false and true.
    //
    setSortBy: function(fieldName) {
      this.set('sortBy', fieldName);
      this.toggleProperty('sortAscending');

      console.log('Sorting by ', fieldName);
      console.log('Sorting Asc?: ', this.get('sortAscending'));

      return false;
    }
  }
});
```

Now we need to call the `setSortBy` action in the **app/templates/friends/index.hbs**

app/templates/friends/index.hbs

```
<table class="primary friends-table">
  <thead>
    <tr>
      <th {{action "setSortBy" "firstName"}}> Name</th>
      <th {{action "setSortBy" "totalArticles"}}>Articles</th>
      <th></th>
    </tr>
  </thead>
  <tbody>
    {{#each friend in model}}
      <tr>
        <td>{{link-to friend.fullName "articles" friend}}</td>
        <td>{{friend.totalArticles}}</td>
        <td><a href="#" {{action "delete" friend}}>delete</a></td>
      </tr>
    {{/each}}
  </tbody>
</table>
```

Let us add add some CSS so we have a cursor on the name and articles rows:

app/styles/app.css

```
.friends-table thead tr {
  cursor: pointer;
}
```

Now If we go to http://localhost:4200/friends and click on **Name** or **Articles**, we'll see that our action is being fired and something like the following logged to the browser's console:

```
Sorting by  firstName
Sorting Asc?:  false
Sorting by  totalArticles
Sorting Asc?:  true
```

But our list is not changing and we don't see the URL changing either, we need to refresh our model every time those values change and also the URL. To achieve this we'll use a useful feature called Query Parameters[102] that allows us to persist application state in the URL as parameters, generating URLs like `/friends?sortBy=name&sortAscending=true`.

[102] http://emberjs.com/guides/routing/query-params/

Query Parameters

To use query parameters we need to specify a property called `queryParams` in the controller associated with this route, and then list every property that should persist as query parameter.

In our scenario we'll modify the controller as follows:

app/controllers/friends/index.js

```
import Ember from 'ember';

export default Ember.Controller.extend({
  queryParams: ['sortBy', 'sortAscending'],
  sortAscending: true,
  sortBy: 'firstName',
  actions: {
    // omitted
  }
```

If we visit http://localhost:4200/friends the URL won't have any query parameters, but as soon as we click any of the headers the query parameters will change. Query parameters are only included when the default value for the property changes. In our case, that would be when `sortAscending` changes to something different from `true` and `sortBy` to something different from `firstName`.

Now we can refresh the browser or copy the URL into a new tab and we'll see the same query parameters, but the data is still not changing, we'll see how to fix that shortly.

We can also use query params with the `link-to` helper. If we want a link to the friends index sorted by `totalArticles`, we can write it like this: `{{#link-to 'friends' (query-params sortBy="totalArticles")}}Friends{{/link-to}}`

Refreshing the model when query parameters changes

By default the model hook won't be called if any of the query parameters change, but there are scenarios where this can be the desired behavior. For example, when we are using pagination, or if we want to do server side sorting, under that scenario we'll ask the API for the users sorted by a given field in ascending or descending order.

Let's use the query parameters to change our friends order, since our API supports `sortBy` and`sortAscending`, we can have the route make a full transition when any of the `queryParams` change. To do this, we'll need to specify a property in the route called `queryParams` where we explicitly mark the parameters that we want to cause a full transition.

app/routes/friends/index.js

```
import Ember from 'ember';

export default Ember.Route.extend({
  queryParams: {
    sortBy: {
      refreshModel: true
    },
    sortAscending: {
      refreshModel: true
    }
  },
  model: function(params) {
    return this.store.findQuery('friend', params);
  }
});
```

Now every time we change `sortBy` or `sortAscending`, the model hook for **app/routes/friends/index.js** will be called, making a request to the API similar to the following and our friends list will be updated accordingly:

```
/api/v4/friends?sortBy=firstName&sortAscending=true
```

Further Reading

Query parameters is one of the best documented features on Ember. We recommend the official guide for more information: http://emberjs.com/guides/routing/query-params/[103].

Tasks

Use query parameters on the articles index to show or hide articles depending on their state. If the query parameter `showReturned` is true, then all the articles are displayed. Otherwise, only the ones in the `borrowed` state are shown.

Tip: We can have a computed property called `filteredResults` on the controller that updates if `showReturned` changes. See also: Ember.Enumerable#filterBy[104].

[103] http://emberjs.com/guides/routing/query-params/
[104] http://emberjs.com/api/classes/Ember.Enumerable.html#method_filterBy

Testing Ember.js applications

In this chapter we'll cover the basics of unit and acceptance testing in Ember.js applications and recommend a couple of resources that can help us expand our knowledge in this area.

Unit Testing

When we run the generators, they create unit test files by default. We can view all the generated unit tests if we go to `tests/unit`:

Unit tests

```
$ ls tests/unit/
adapters        controllers        models                utils
components        helpers                routes
```

Tests are automatically grouped by type. If we open the unit test for our friend model, we'll see the following:

tests/unit/models/friend-test.js

```
import { test, moduleForModel } from 'ember-qunit';

moduleForModel('friend', 'Friend', { needs: ['model:article'] });

test('it exists', function(assert) {
  var model = this.subject();
  assert.ok(model);
});
```

At the beginning of the test we import a set of helpers from ember-qunit[105], which is a library that wraps a bunch of functions to facilitate testing with **QUnit**.

`moduleForModel` received the name of the model we are testing, a description, and some options. In our scenario, we specify that the tests need a model called **article** because of the existing relationship between them.

[105] https://github.com/rwjblue/ember-qunit

Next, the test includes a basic assertion that the model exists. `this.subject()` would be an instance of a `friend`.

We have two ways of running tests. The first one is via the browser while we run the development server. We can navigate to http://localhost:4200/tests[106] and our tests will be run. The second method is using a tests runner. At the moment **ember-cli** has built-in support for **Testem** with PhantomJS[107], which we can use to run our tests on a CI server. To run tests in this mode, we only need to do `ember test`.

We can also run tests with the command `npm test` which is aliased to `ember test` in `package.json`.

Let's write two more tests for our friend model. We want to check that the computed property `fullName` behaves as expected and that the relationship articles is properly set.

tests/unit/models/friend-test.js

```
import { test, moduleForModel } from 'ember-qunit';
import Ember from 'ember';

moduleForModel('friend', 'Friend', {
  needs: ['model:article']
});

test('it exists', function(assert) {
  var model = this.subject();
  assert.ok(model);
});

test('fullName joins first and last name', function(assert) {
  var model = this.subject({firstName: 'Syd', lastName: 'Barrett'});

  assert.equal(model.get('fullName'), 'Syd Barrett');

  Ember.run(function() {
    model.set('firstName', 'Geddy');
  });

  assert.equal(model.get('fullName'), 'Geddy Barrett', 'Updates fullName');
});
```

[106] http://localhost:4200/tests
[107] http://phantomjs.org/

```
test('articles relationship', function(assert) {
  var klass  = this.subject({}).constructor;

  var relationship = Ember.get(klass, 'relationshipsByName').get('articles');

  assert.equal(relationship.key, 'articles');
  assert.equal(relationship.kind, 'hasMany');
});
```

We can run our tests by going directly to the following URL: http://localhost:4200/tests?module=Friend[108].

The first test verifies that `fullName` is calculated correctly. We have to wrap `model.set('firstName', 'Geddy');` in `Ember.run` because it has an asynchronous behavior. If we modify the implementation for `fullName` such that it doesn't return first and last names, the tests will fail.

The second test checks that we have set up the proper relationship to `articles`. Something similar could go in the articles model tests. If we call `constructor` on an instance to a model, that will give us access to the class of which it is an instance.

Let's add other unit test for `app/utils/date-helpers`:

tests/unit/utils/date-helpers-test.js

```
import { module, test } from 'ember-qunit';
import dateHelpers from '../../../utils/date-helpers';

module('Utils: formatDate');

test('formats a date object', function(assert) {
  var date = new Date("11-3-2015");
  var result = dateHelpers.formatDate(date, 'ddd MMM DD YYYY');

  assert.equal(result, 'Mon Nov 03 2014', 'returns a readable string');
});
```

We import the function we want to test and then check that it returns the date as a readable string. We can run the test by going to http://localhost:4200/tests?module=Utils%3A%20formatDate[109].

Acceptance Tests

With acceptance tests we can verify workflows in our application. For example, making sure that we can add a new friend, that if we visit the friend index a list is rendered, etc. An acceptance test basically emulates a real user's experience of our application.

[108] http://localhost:4200/tests?module=Friend
[109] http://localhost:4200/tests?module=Utils%3A%20formatDate

Ember has a set of helpers to simplify writing these kinds of tests. There are synchronous[110] and asynchronous[111] helpers. We use the former for tests that don't have any kind of side-effect, such as checking if an element is present on a page, and the latter for tests that fire some kind of side-effect. For example, clicking a link or saving a model.

Let's write an acceptance test to verify that we can add new friends to our application. We can generate an acceptance test with the generator **acceptance-test**.

```
$ ember g acceptance-test friends/new
installing
  create tests/acceptance/friends/new-test.js
```

If we visit the generated test, we'll see the following:

tests/acceptance/friends/new-test.js

```
import Ember from 'ember';
import { module, test } from 'ember-qunit';
import startApp from '../helpers/start-app';

var application;

module('Acceptance: FriendsNew', {
  beforeEach: function() {
    application = startApp();
  },
  afterEach: function() {
    Ember.run(application, 'destroy');
  }
});

test('visiting /friends/new', function(assert) {
  visit('/friends/new');

  andThen(function() {
    assert.equal(currentPath(), 'friends/new');
  });
});
```

We need to replace `import startApp from '../helpers/start-app';` with `import startApp from '../../helpers/start-app';` and then make the assertion of `currentPath` look for `friends.new` instead of `friends/new`.

[110] http://emberjs.com/guides/testing/test-helpers/#toc_wait-helpers
[111] http://emberjs.com/guides/testing/test-helpers/#toc_asynchronous-helpers

Now we can run our tests by visiting http://localhost:4200/tests[112] or, if we want to run only the acceptance tests for Friends New, http://localhost:4200/tests?module=Acceptance%3A%20FriendsNew[113].

Let's add two more tests but this time starting from the index URL. We want to validate that we can navigate to new and then check that it redirects to the correct place after creating a new user.

Tests new friend: tests/acceptance/friends/new-test.js

```
test('Creating a new friend', function(assert) {
  visit('/');
  click('a[href="/friends/new"]');
  andThen(function() {
    assert.equal(currentPath(), 'friends.new');
  });
  fillIn('input[placeholder="First Name"]', 'Johnny');
  fillIn('input[placeholder="Last Name"]', 'Cash');
  fillIn('input[placeholder="email"]', 'j@cash.com');
  fillIn('input[placeholder="twitter"]', 'jcash');
  click('input[value="Save"]');

  //
  // Clicking save will fire an async event.
  // We can use andThen, which will be called once the promises above
  // have been resolved.
  //

  andThen(function() {
    assert.equal(
      currentRouteName(),
      'friends.show.index',
      'Redirects to friends.show after create'
    );
  });

});
```

The second test we want to add checks that the application stays on the new page if we click save, without adding any fields, and that an error message is displayed:

[112] http://localhost:4200/tests

[113] http://localhost:4200/tests?module=Acceptance%3A%20FriendsNew

Tests new friend: tests/acceptance/friends/new-test.js

```
test('Clicking save without filling fields', function(assert) {
  visit('/friends/new');
  click('input[value="Save"]');
  andThen(function() {
    assert.equal(
      currentRouteName(),
      'friends.new',
      'Stays on new page'
    );
    assert.equal(
      find("h2:contains(You have to fill all the fields)").length,
      1,
      "Displays error message"
    );
  });

});
```

Mocking the API response

On the previous tests we hit the API, but this is not a common scenario. Normally we'd like to mock the interactions with the API. To do so we have different alternatives. One is to use Pretender[114], a library that allows us to mock requests with a simple DSL.

Another alternative is to use the built-in mock generator[115] in **ember-cli**. This basically takes advantage of the Express server used for development and extends it to capture requests to our API end-points. With this tool, we can control what we would like to return for each request.

Let's create a mock for `api/articles`:

```
$ ember g http-mock articles
installing
  create server/.jshintrc
  create server/index.js
  create server/mocks/articles.js
  install package connect-restreamer
```

If we open the generated file `server/mocks/articles.js`, we'll see the following:

[114] https://github.com/trek/pretender
[115] http://www.ember-cli.com/#mocks-and-fixtures

server/mocks/articles.js

```
module.exports = function(app) {
  var express = require('express');
  var articlesRouter = express.Router();
  articlesRouter.get('/', function(req, res) {
    res.send({"articles":[]});
  });
  app.use('/api/articles', articlesRouter);
};
```

This intercepts the call to any request starting with `/api/articles`. If it is a GET to `/`, it will return `{"articles":[]}`.

Suppose we want to mock the request for a particular article. We can add the following:

server/mocks/articles.js

```
module.exports = function(app) {
  var express = require('express');
  var articlesRouter = express.Router();
  articlesRouter.get('/', function(req, res) {
    res.send({"articles":[]});
  });
  articlesRouter.get('/articles/74', function(req, res) {
    res.send({
      "article":{
        "id":74,
        "created_at":"2014-11-03T21:30:47.869Z",
        "description":"foo",
        "state":"borrowed",
        "notes":"bar",
        "friend_id":153
      }
    });
  });
  app.use('/api/articles', articlesRouter);
};
```

This will intercept any GET request to `/articles/74` and return the mocked article.

Further Reading

During EmberConf 2014, Eric Berry[116] gave a great talk called The Unofficial, Official Ember Testing Guide[117] where he walked us through testing in Ember.js. Eric also contributed an excellent guide for testing that is now the official guide on the Ember.js website. We recommend the official guide, which provides a complete overview from unit to acceptance testing: http://emberjs.com/guides/testing/[118].

To know more about using mocks and fixtures, we recommend the following presentation: Real World Fixtures[119] by Chris Ball[120].

[116] https://twitter.com/coderberry
[117] http://www.confreaks.com/videos/3310-emberconf2014-the-unofficial-official-ember-testing-guide
[118] http://emberjs.com/guides/testing
[119] https://speakerdeck.com/cball/real-world-fixtures
[120] https://twitter.com/cball_

PODS

Until now we have organized our project files by type, so we have all the models under `app/models`, controllers under `app/controllers`, and so on.

As we mentioned in the section on adapters, **ember-cli** allows us to group things that are logically related under a single directory. Such a structure is known as "pods".

The following shows us how the resolver tries to find the friend adapter:

Resolving the friend adapter

```
[ ] adapter:friend .............borrowers/friend/adapter
[ ] adapter:friend .............undefined
[ ] adapter:friend .............borrowers/adapters/friend
[ ] adapter:friend .............undefined
```

First it tries to find the module `adapter` under the namespace `friend` and then moves to the namespace `adapters`.

We are currently able to structure our projects using pods or by grouping items by their type, but the way forward is to start using pods. Ember 2.0 introduces the concept of Routeable Components, and it will expect us to place some files following the pod convention.

For changes coming in Ember 2.0, read: The Road to Ember 2.0 RFC[121]

Using pods

Let's change our routes, controllers, and templates related to a friend so that they are located in the pod called `app/friends`.

One easy way to find out where we should place our files is to look at the resolver log. We can enable it by setting the property `ENV.APP.LOG_RESOLVER` to `true` in `app/environment.js`.

The following is the lookup log for objects related to a friend:

[121] https://github.com/emberjs/rfcs/pull/15

pods lookup

```
[ ] template:friends ............ borrowers/friends/template

[ ] route:friends/index .......... borrowers/friends/index/route
[ ] controller:friends/index ..... borrowers/friends/index/controller
[ ] template:friends/index ....... borrowers/friends/index/template

[ ] route:friends/new ............ borrowers/friends/new/route
[ ] controller:friends/new ....... borrowers/friends/new/controller
[ ] template:friends/new ......... borrowers/friends/new/template

[ ] route:friends/show ........... borrowers/friends/show/route
[ ] controller:friends/show ...... borrowers/friends/show/controller
[ ] template:friends/show ........ borrowers/friends/show/template
```

We can start by creating a directory called friends followed by the child directories new, edit, index, and show.

```
$ mkdir app/friends
$ mkdir app/friends/new
$ mkdir app/friends/show
$ mkdir app/friends/index
$ mkdir app/friends/edit
```

With the directories in place, we can start by moving the routes:

```
$ mv app/routes/friends/index.js app/friends/index/route.js
$ mv app/routes/friends/show.js app/friends/show/route.js
$ mv app/routes/friends/edit.js app/friends/edit/route.js
$ mv app/routes/friends/new.js app/friends/new/route.js
$ rm -rf app/routes/friends
```

If we run our acceptance tests for friends http://localhost:4200/tests?module=Acceptance%3A%20FriendsNew[122], everything should work.

Next let's move the templates:

[122] http://localhost:4200/tests?module=Acceptance%3A%20FriendsNew

```
$ mv app/templates/friends/index.hbs app/friends/index/template.hbs
$ mv app/templates/friends/show.hbs app/friends/show/template.hbs
$ mv app/templates/friends/edit.hbs app/friends/edit/template.hbs
$ mv app/templates/friends/new.hbs app/friends/new/template.hbs
```

On the edit and new template, we are using a partial called "form." The pods loookup for partials expects the template to be located in the following file: `borrowers/friends/-form/template`. Let's move it there:

```
$ mkdir app/friends/-form
$ mv app/templates/friends/-form.hbs app/friends/-form/template.hbs
$ rm -rf app/templates/friends
```

Now the controllers:

```
$ mv app/controllers/friends/index.js app/friends/index/controller.js
$ mv app/controllers/friends/edit.js app/friends/edit/controller.js
$ mv app/controllers/friends/new.js app/friends/new/controller.js
$ mv app/controllers/friends/base.js app/friends/base-controller.js
```

Notice that we moved the base controller to `app/friends/base-controller.js`. We need to update the references in `app/friends/edit/controller.js` and `app/friends/new/controller.js`.

Instead of:

```
import FriendsBaseController from './base';
```

We need:

```
import FriendsBaseController from '../base-controller';
```

If we update the browser, everything should work. We are now using pods for our friends.

Tasks

Change the structure for articles so that everything is under the pod `articles`.

Deploying Ember.js applications

In this chapter we'll explore different alternatives to deploy our Ember.js applications. We'll talk about S3 and Divshot based deployments where our application is completely separated from our API. Then we'll cover how to do a deployment on Heroku using the `heroku-buildpack-ember-cli`, which allows us to proxy requests to our API. Finally, we'll talk about Redis based deployments in Ruby on Rails and Node.js.

Deploying to S3

In order to host our application in S3, we'll need to change our application adapter so it hits our CORS enabled API and then generate a production build.

To consume the API without using **ember-cli**'s proxy feature, we need to set the property `host` in the application adapter.

To do so, let's add a configuration property called `host` in `config/environment.js` and then read it from there.

Adding host to config/environment.js

```
/* jshint node: true */

module.exports = function(environment) {
  var ENV = {
    host: 'http://api.ember-cli-101.com',
    // ...
```

Now we can use it in the application adapter as follows:

app/adapters/application.js

```
import DS from 'ember-data';
import config from '../config/environment';

export default DS.ActiveModelAdapter.extend({
  host: config.host,
  namespace: 'api/v4',
  coalesceFindRequests: true
});
```

We also need to change `app/routes/index.js` to use the host:

app/routes/index.js

```
import Ember from 'ember';
import request from 'ic-ajax';
import config from '../config/environment';

export default Ember.Route.extend({
  model: function()  {
    var host = config.host || '';

    return request(host + '/api/friends').then(function(data){
      return {
        friendsCount: data.friends.length
      };
    });
  }
});
```

Now we can stop the server and run it again without the option `--proxy`.

Next we need to generate the production build using the command `ember build`.

When we run `ember server`, we always run a `build` and add some extra stuff so that we can run our project in development, but we don't need the same files in production.

When we do `ember build`, the output goes by default to the directory `dist`. Let's check that:

ember build

```
borrowers $ ember build
version: 0.1.9
Building...
Built project successfully. Stored in "dist/".
```

Inspecting the `dist` directory, we'll see the following contents:

```
|- assets
|-- |- borrowers.css
|-- |- borrowers.js
|-- |- failed.png
|-- |- passed.png
|-- |- test-loader.js
|-- |- test-support.css
|-- |- test-support.js
|-- |- vendor.css
|-- |- vendor.js
|- crossdomain.xml
|- font
|-- |- fontello.eot
|-- |- fontello.svg
|-- |- fontello.ttf
|-- |- fontello.woff
|- index.html
|- robots.txt
|- testem.js
|- tests
    |- index.html
```

Remember we can see the options for a command passing the option `--help` like `ember build --help`.

Let's talk about the assets directory first. All our JavaScript and stylesheet files will end in this directory. We can also put other kinds of assets, such as images or fonts, under `public/assets` and they will be merged into this directory. If we had the image `public/assets/images/foo.png` we could reference it in our stylesheets like `images/foo.png`.

What about those test files? They are used for testing and only included in development or test environments. If we go to http://localhost:4200/tests[123] and inspect the network tab, we'll see that those files are being used.

The tests directory is the entry point for running tests. `testem.js` is used by default when we do `ember test`. It uses **Testem** to run the test with **PhantomJS**.

If we run the build command but we specify production environment (e.g., `ember build --environment production`) we'll see a very different output:

[123] http://localhost:4200/tests

```
.
|-- assets
|- |-- borrowers-97a85d25222a06c4a39d475c7ad27a73.js
|- |-- borrowers-985aabef341eea2a8b20d3e9e685d6b0.css
|- |-- images
|- |-- vendor-9877b53c34630081b26b7b9fd19d4bb8.css
|- |-- vendor-b29ae2f2e402c33a5d9c683aac4e0f8e.js
|-- crossdomain.xml
|-- font
|- |-- fontello.eot
|- |-- fontello.svg
|- |-- fontello.ttf
|- |-- fontello.woff
|-- index.html
|-- robots.txt
```

We have fewer files this time. Nothing related with testing is included because that is only a development/tests concern. Our assets files were fingerprinted and minified. If we open `dist/index.html` we'll see that the references to them were updated as well:

```
<link rel="stylesheet" href="assets/vendor-9877b53c34630081b26b7b9fd19d4bb8.css">
<link rel="stylesheet" href="assets/borrowers-985aabef341eea2a8b20d3e9e685d6b0.c\
ss">
```

Fingerprinting is achieved using broccoli-asset-rev[124]. This allows us the option to select the format of the files we want to fingerprint and to append an URL to every asset.

All our assets should ideally be kept under the directory `/assets`, so let's make sure our fonts are put in there as well. To do this, we need to modify our `Brocfile` and the references to the fonts in `vendor/fontello/fontello.css`.

To accomplish the first part we simply need to specify `assets/fonts` as the `destDir` for our imported fonts:

[124] https://github.com/rickharrison/broccoli-asset-rev

Brocfile.js

```
app.import('vendor/fontello/font/fontello.ttf', {
  destDir: 'assets/fonts'
});
app.import('vendor/fontello/font/fontello.eot', {
  destDir: 'assets/fonts'
});
app.import('vendor/fontello/font/fontello.svg', {
  destDir: 'assets/fonts'
});
app.import('vendor/fontello/font/fontello.woff', {
  destDir: 'assets/fonts'
});
```

If we run `ember build --environment production`, we'll find our fonts under `assets/fonts`.

Putting fonts under assets directory

```
.
|-- assets
|-- |-- borrowers-985aabef341eea2a8b20d3e9e685d6b0.css
|-- |-- borrowers-da3abd96a2852e1cfa758c2d41b82a5e.js
|-- |-- fonts
|-- |-- |-- fontello.eot
|-- |-- |-- fontello.svg
|-- |-- |-- fontello.ttf
|-- |-- |-- fontello.woff
|-- |-- images
|-- |-- vendor-9877b53c34630081b26b7b9fd19d4bb8.css
|-- |-- vendor-b29ae2f2e402c33a5d9c683aac4e0f8e.js
|-- crossdomain.xml
|-- index.html
|-- robots.txt
```

Next we need to replace `vendor/fontello/fontello.css` to reference the fonts relative to `fonts/` instead of `../font`:

vendor/fontello/fontello.css

```
@font-face {
  font-family: 'fontello';
  src: url('fonts/fontello.eot?59907090');
  src: url('fonts/fontello.eot?59907090#iefix') format('embedded-opentype'),
       url('fonts/fontello.woff?59907090') format('woff'),
       url('fonts/fontello.ttf?59907090') format('truetype'),
       url('fonts/fontello.svg?59907090#fontello') format('svg');
  font-weight: normal;
  font-style: normal;
}
```

Now we are ready to deploy to an S3 bucket. We need to create the bucket and enable static website hosting. Let's set up an index document, `index.html`.

The following guide explains how to set up your S3 bucket: Hosting a Static Website on Amazon S3[125]

Once the bucket is set up, we can run `ember build --environment production` and then manually upload all the files under `dist`. The following is an example of the site working on S3: http://ember-cli-101.s3-website-us-east-1.amazonaws.com/[126]

It is very important that we set our bucket as public. To do this, we can use the following bucket policy:

S3 policy

```
{
        "Version": "2012-10-17",
        "Statement": [
                {
                        "Sid": "AddPerm",
                        "Effect": "Allow",
                        "Principal": "*",
                        "Action": "s3:GetObject",
                        "Resource": "arn:aws:s3:::REPLACE-WITH-REAL-BUCKET-NAME/*"
                }
        ]
}
```

[125] http://docs.aws.amazon.com/AmazonS3/latest/dev/WebsiteHosting.html
[126] http://ember-cli-101.s3-website-us-east-1.amazonaws.com/

The following tutorial explains how to achieve a setup using custom routing and Cloudfront: Hosting a Static Website on Amazon Web Services[127]

If we decide to use Cloudfront, we need to prepend the URL to our assets. To do this, we simply pass the option in the Brocfile as follows:

Brocfile.js

```
var app = new EmberApp({
  fingerprint: {
    prepend: 'https://d29sqib8gy.cloudfront.net/'
  },
});
```

If we run `ember build --environment production` and open `dist/index.html`, we'll notice the URL in our assets.

```
<script src="https://d29sqib8gy.cloudfront.net/assets/vendor-b29ae2f2e402c33a5d9\
c683aac4e0f8e.js"></script>
<script src="https://d29sqib8gy.cloudfront.net/assets/borrowers-c459411ce1cc8332\
ef795be81d96d1b6.js"></script>
```

A better approach to uploading our files to S3 is to create a task to do this for us. At the moment there is no built-in support for this in ember-cli, but we can use Grunt[128] with the following plugin: grunt-aws-s3[129].

Deploying to Divshot

Divshot[130] is a PaaS for deploying static websites. This is probably the easiest way to deploy such applications, and Robert Jackson[131] wrote an **ember-cli addon** to make it even easier to deploy our ember-cli applications.

Before installing the addon, we have to first create an account with them and then install their command line interface:

```
npm install -g divshot-cli
```

After installing **divshot-cli**, we need to login typing `divshot login`.

Once we are logged in, we are ready to deploy our application. First install the addon ember-cli-divshot[132]:

[127] http://docs.aws.amazon.com/gettingstarted/latest/swh/website-hosting-intro.html
[128] http://gruntjs.com/
[129] https://github.com/MathieuLoutre/grunt-aws-s3
[130] https://divshot.com/
[131] https://twitter.com/rwjblue
[132] https://github.com/rwjblue/ember-cli-divshot

```
npm install ember-cli-divshot --save-dev
```

With the addon installed, we need to set up DivShot with `ember generate divshot`, and after that we can deploy just running `ember divshot push`.

Deploying to DivShot

```
$ ember divshot push
version: 0.1.9
Built project successfully. Stored in "dist".
Creating build ...
Hashing Directory Contents ...
Synced!
Finalizing build ...
Releasing build to development ...
Success: Application deployed to development
Success: You can view your app at: http://development.borrowers.divshot.io
```

That's it! Our application has been deployed to http://development.borrowers.divshot.io[133].

Deploying to Heroku with the heroku-buildpack-ember-cli

Deploying to Heroku[134] is a simple process thanks to Tony Coconate[135]'s heroku-buildpack-ember-cli[136].

Assuming we have already created an account on Heroku and installed heroku toolbelt[137], we can now deploy with the following steps.

First we need to create an application based on the buildpack:

[133] http://development.borrowers.divshot.io
[134] http://heroku.com/
[135] https://twitter.com/tonycoco
[136] https://github.com/tonycoco/heroku-buildpack-ember-cli
[137] https://toolbelt.heroku.com/

```
$ heroku create --buildpack https://github.com/tonycoco/heroku-buildpack-ember-c\
li.git
Creating polar-cove-8298... done, stack is cedar
BUILDPACK_URL=https://github.com/tonycoco/heroku-buildpack-ember-cli.git
https://polar-cove-8298.herokuapp.com/ | git@heroku.com:polar-cove-8298.git
Git remote heroku added
```

Now we can deploy doing `git push heroku master`. We can see our application running on Heroku: http://polar-cove-8298.herokuapp.com/[138]

Using the Proxy Feature.

Supposing we don't want to enable CORS in our API, the build-pack has a **Proxy** feature that acts similarly to the one included with **ember-cli**.

Using the following command, we can set up the URL to which we want to proxy our request:

```
heroku config:set API_URL=http://api.ember-cli-101.com/
```

We can find more info about this in the Github repository[139].

ember-cli-deploy

During EmberConf 2015, Luke Melia[140] gave a talk called The Art of Ember App Deployment[141].

Luke presented a solution to keep the deployment of JavaScript applications separate from the backend. The basic idea is to deploy our assets to a CDN and then pass the generated `index.html` via Redis to the application serving it.

During this talk Luke announced the creation of a "community supported" addon for "Lightning Fast Deployments of Ember-CLI Apps" which is called ember-cli-deploy[142], it makes it super easy to implement Luke's ideas.

[138] http://polar-cove-8298.herokuapp.com/
[139] https://github.com/tonycoco/heroku-buildpack-ember-cli#api-proxy
[140] https://twitter.com/lukemelia
[141] https://www.youtube.com/watch?v=4EDetv_Rw5U
[142] https://github.com/ember-cli/ember-cli-deploy

Updating your project to the latest version of ember-cli

ember-cli is a project that is still moving quickly, so from time to time we'll need to update our applications to use the latest version.

By the time this chapter was written the application was using ember-cli **0.0.46**, which was one of the last releases before moving to **0.1.X**. Now we want to move to the newest version in npm.

The following steps are the same that come listed with every release of ember-cli:

1. We get rid of the current installed version: **npm uninstall -g ember-cli**
2. A lot of libraries that ember-cli relied on were updated as well. We want to make sure we are getting the latest versions from npm and not using the ones we had previously installed. To do that, we clean the npm cache with: **npm cache clean**
3. Now we'll do the same with bower: **bower cache clean**
4. Finally, we'll install ember-cli again: **npm install -g ember-cli**

Once we have installed the latest version of **ember-cli**, we need to update our project. Let's run the following commands in the borrowers app directory:

1. Remove installed libraries, dist files, and temporary files: rm -rf node_modules bower_components dist tmp
2. Next we need to update **ember-cli**'s version in **package.json** running: npm install –save-dev ember-cli
3. Install dependencies: **npm install && bower install**

We are almost done. We have upgraded **ember-cli** and **dependencies** successfully, but we still need to upgrade some files in our projects. The good news is that we don't have to do it all manually. We can use the command **ember init**. When we run this it will try to make some changes in some of our existing files, and we can answer these requests with any of the following options:

Updating ember-cli

```
  y) Yes, overwrite
  n) No, skip
  d) Diff
  h) Help, list all options
```

A good approach is to first inspect what changed with the option **d** and then decide if we want to accept the change or not.

Let's run **ember init** and see the output. We'll also include some comments that are not part of the original output just to clarify:

```
$ ember init .
version: 0.1.9
installing

#
# ember-cli will try to replace some files with their blueprint version
# we'll respond with d to see the diff
#

[?] Overwrite /borrowers/Brocfile.js? (Yndh) d

--- /borrowers/Brocfile.js
+++ /borrowers/Brocfile.js
@@ -3,28 +3,18 @@
 var EmberApp = require('ember-cli/lib/broccoli/ember-app');

 var app = new EmberApp();

+// Use `app.import` to add additional libraries to the generated
+// output files.
+//
+// If you need to use different assets in different
+// environments, specify an object as the first parameter. That
+// object's keys should be the environment name and the values
+// should be the asset to use in that environment.
+//
+// If the library that you are including contains AMD or ES6
+// modules that you would like to import into your application,
```

```
+// please specify an object with the list of modules as keys
+// along with the exports of each module as its value.
-app.import('vendor/fontello/fontello.css');
-app.import('vendor/fontello/font/fontello.ttf', {
-  destDir: 'font'
-});
-app.import('vendor/fontello/font/fontello.eot', {
-  destDir: 'font'
-});
-app.import('vendor/fontello/font/fontello.svg', {
-  destDir: 'font'
-});
-app.import('vendor/fontello/font/fontello.woff', {
-  destDir: 'font'
-});
-app.import('bower_components/picnic/releases/picninc.min.css);
-app.import('bower_components/moment/moment.js');
-app.import('bower_components/borrowers-dates/index.js', {
-  exports: {
-    'borrowers-dates': [
-      'format'
-    ]
-  }
-});

 module.exports = app.toTree();

[?] Overwrite /borrowers/Brocfile.js? n
```

In the diff above, the line with a plus (+) sign is what will get added and the lines with the minus (-) will be removed.

In this specific scenario, nothing important got added to the Brocfile and it is trying to remove our imports. We can simply ignore this file with the option **n**.

Next it asks us if we want to overwrite the README. This file is only a blueprint, so we can ignore it safely by hitting the key **n**.

```
[?] Overwrite /borrowers/README.md? (Yndh)
```

The next file is **index.html**. This file can change from time to time, so we should keep an eye on it. Let's see the diff with **d**:

```
[?] Overwrite /borrowers/app/index.html? d

--- /borrowers/app/index.html
+++ /borrowers/app/index.html
@@ -6,20 +6,14 @@
     <title>Borrowers</title>
     <meta name="description" content="">
     <meta name="viewport" content="width=device-width, initial-scale=1">

+    {{content-for 'head'}}
-    {{BASE_TAG}}

     <link rel="stylesheet" href="assets/vendor.css">
     <link rel="stylesheet" href="assets/borrowers.css">
   </head>
   <body>
-    <script type="text/javascript">
-      window.EmberENV = {{EMBER_ENV}};
-    </script>
     <script src="assets/vendor.js"></script>
     <script src="assets/borrowers.js"></script>
-    <script type="text/javascript">
-      window.Borrowers = require('borrowers/app')['default'].create({{APP_CONFI\
G}});
-    </script>
   </body>
 </html>


[?] Overwrite /borrowers/app/index.html? (Yndh) Y
```

At the beginning, ember-cli used to include inline scripts for starting the app and defining a global ENV variable. It has changed to encourage users to write CSP-compliant applications.

CSP (Content Security Policy) is basically a mechanism to help us write more secure applications. The following is a great write-up by the HTML5 Rocks folks: An Introduction to Content Security Policy[143].

Next we have **router.js** and **application.hbs**. We won't include the output for the diffs for the sake of brevity, but the first one doesn't change very often and the latter one can be ignored since we don't want any changes in our application template.

[143] http://www.html5rocks.com/en/tutorials/security/content-security-policy/

Unless we are in a version lower than **0.0.46**, we can safely ignore both.

lang=bash

```
[?] Overwrite /borrowers/app/router.js? (Yndh) n
[?] Overwrite /borrowers/app/templates/application.hbs? (Yndh) n
```

Next it will ask us if we want to overwrite bower.json. This and package.json will probably change often since dependencies get updated frequently. Here is where our revision control system plus a bit of strategy comes in really handy. Let's inspect the diff with **d**:

lang=bash

```
[?] Overwrite /borrowers/bower.json? (Yndh) d

--- /borrowers/bower.json
+++ /borrowers/bower.json
@@ -2,19 +2,16 @@
   "name": "borrowers",
   "dependencies": {
     "handlebars": "~1.3.0",
     "query": "^1.11.1",
     "ember": "1.11.0",
     "ember-data": "1.0.0-beta.10",
     "ember-resolver": "~0.1.7",
+    "loader.js": "stefanpenner/loader.js#1.0.1",
-    "loader": "stefanpenner/loader.js#1.0.1",
     "ember-cli-shims": "stefanpenner/ember-cli-shims#0.0.3",
     "ember-cli-test-loader": "rwjblue/ember-cli-test-loader#0.0.4",
     "ember-load-initializers": "stefanpenner/ember-load-initializers#0.0.2",
     "ember-qunit": "0.1.8",
     "ember-qunit-notifications": "0.0.4",
+    "qunit": "~1.15.0"
-    "qunit": "~1.15.0",
-    "picnic": "https://github.com/picnicss/picnic.git",
-    "moment": "~2.8.3",
-    "borrowers-dates": "~0.0.1"
   }
 }

[?] Overwrite /borrowers/bower.json? (Yndh) Y
```

In this scenario Ember and Ember-Data didn't change but it might happen from time to time that Ember and Ember-Data get updated to their latest available release, in case we don't want to update those libraries we can just ignore the changes to those lines.

We responded with yes to the previous command. In this particular case, not many dependencies changed but we need the update anyways. We also notice that the dependencies we added were deleted.

How do we deal with this in a scenario where there are a lot of dependencies changed and the ones introduced by us get deleted? Version control systems to the rescue!

A good strategy is to put all of our dependencies at the end of the default libraries (after QUnit) and then simply overwrite the whole file when updating.

If we are using Git, we can bring back that last hunk that was deleted from our file; it can easily be done with any GUI based tool. If you are an Emacs user you can use diff-hl-mode, Sublime emacs-git-gutter, or Vim's vim-gitgutter.

Next is **environment.js**. We should check the changes here since it could have breaking changes:

```
[?] Overwrite /borrowers/config/environment.js? d

--- /borrowers/config/environment.js
+++ /borrowers/config/environment.js
@@ -20,9 +20,9 @@
   };

   if (environment === 'development') {
     // ENV.APP.LOG_RESOLVER = true;
+    ENV.APP.LOG_ACTIVE_GENERATION = true;
-    // ENV.APP.LOG_ACTIVE_GENERATION = true;
     // ENV.APP.LOG_TRANSITIONS = true;
     // ENV.APP.LOG_TRANSITIONS_INTERNAL = true;
     ENV.APP.LOG_VIEW_LOOKUPS = true;
     }
```

As in previous scenarios, there are not many significant changes, so we can simply ignore by responding with **n**.

Next is **package.json**. Most of the changes are packages being updated. We'll say yes to this change. Again, use the same strategy mentioned with **bower.json** by putting our own libraries at the end.

```
[?] Overwrite /borrowers/package.json? (Yndh) d

--- /borrowers/package.json
+++ /borrowers/package.json
@@ -18,13 +18,14 @@
   "author": "",
   "license": "MIT",
   "devDependencies": {
     "body-parser": "^1.2.0",
+    "broccoli-asset-rev": "0.3.0",
-    "broccoli-asset-rev": "0.1.1",
     "broccoli-ember-hbs-template-compiler": "^1.6.1",
+    "ember-cli": "0.1.1",
+    "ember-cli-content-security-policy": "0.2.0",
-    "ember-cli": "^0.1.1",
     "ember-cli-ic-ajax": "0.1.1",
+    "ember-cli-inject-live-reload": "^1.2.2",
-    "ember-cli-inject-live-reload": "^1.0.2",
     "ember-cli-qunit": "0.1.0",
     "ember-data": "1.0.0-beta.10",
     "express": "^4.8.5",
     "glob": "^4.0.5"
```

Next is **.jshintrc**. Ideally we shouldn't have a lot of things in there, but we should be especially careful if we are whitelisting some vars. In this case we'll accept the change because we don't have anything custom.

```
[?] Overwrite /borrowers/tests/.jshintrc? (Yndh)  Y
```

Next we have test helper files, which we are going to accept because we haven't edited any of those files.

```
[?] Overwrite /borrowers/tests/helpers/start-app.js? y
[?] Overwrite /borrowers/tests/index.html? y
[?] Overwrite /borrowers/tests/test-helper.js? y
```

We are almost done. Using the strategy we mentioned to bring back different hunks in a file, we'll make sure **bower.json** has the following packages after **QUnit**:

```
"picnic": "https://github.com/picnicss/picnic.git",
"moment": "~2.8.3",
"borrowers-dates": "~0.0.1"
```

Now we are done. If we run **ember server –proxy http://api.ember-cli-101.com**, the application should start without any problems.