ember 101

```
adolfo@terminus-2 ~/code/101/borrowers $ tree . -L 1
.
├── Brocfile.js
├── README.md
├── app
├── bower.json
├── bower_components
├── config
├── dist
├── node_modules
├── package.json
├── public
├── testem.json
├── tests
├── tmp
└── vendor

9 directories, 5 files
adolfo@terminus-2 ~/code/101/borrowers $
```

by **Adolfo Builes**

ember 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 2016-06-28

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

What you need to know about this book

This book is a beginners guide to the JavaScript framework Ember.

Audience: Programmers with basic JavaScript knowledge.

In order to understand this book, you should have some basic JavaScript knowledge. If you don't, check out the book "Speaking JavaScript"[1] by Axel Rauschmayer, it's free online.

Why should I read this book?

This book covers the basic features to help you build Ember applications. It is written in an informal style and tries to show you common pitfalls to then teach you how to solve those problems and why they are bad ideas.

How to read this book

The book should be read sequentially since every chapter depends on features built or introduced in previous chapters.

Can I learn Ember 2.0 with this book?

Yes, this book will help you learn the current version of Ember and will be updated to reflect future changes (without extra cost), just make sure you enable email notifications from leanpub.

But you are still talking about controllers!

If you are following Ember development, you might know controllers are going away being replaced by something called `Routable Components`. This book talks about controllers but just the necessary stuff so you can use features that still depend on them, once `Routable Components` are official, the book will be updated to reflect those changes.

[1]http://speakingjs.com/

Ember in 2016.

This book is being updated with features from Ember and best practices in 2016. Is still a work in progress, but if you enable notifications from Leanpub or follow me on twitter @abuiles, you'll get the notification when a new version is release.

You can follow the changes in the following pull request https://github.com/abuiles/ember-101/pull/65.

The upcoming version will use JSON API following the example from JSON API By Example[2], remove PODS, talk about the upcoming structure for projects and give more importance to the use of components and how to avoid controllers.

Demo code on GitHub

The repository borrowers-2016[3] contains the code for the application built in the book.

Reporting Errata - Suggesting Edits.

This book is licensed under creative commons, you can report errors or suggest edits in the book's repository ember-101[4].

I hope you enjoy the book!

<3 Adolfo Builes (a.k.a @abuiles).

[2]https://leanpub.com/json-api-by-example
[3]https://github.com/abuiles/borrowers-2016
[4]https://github.com/abuiles/ember-101

Why Ember?

Ember is an evolving JavaScript framework for creating "ambitious web applications", it tries to maximize developers' productivity using a set of conventions in a way that they don't need to think about common idioms when building web applications.

Using Ember in conjuntion with Ember CLI, allows you to reduce big amounts of glue code, giving you the opportunity to focus on what is most important for you: building your application.

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`[5], waiting time is reduced while your assets compile. `QUnit`[6] allows you to write tests, which can then be run with `Testem`[7]. 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)[8]. 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 we 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.

[5] https://github.com/broccolijs/broccoli
[6] http://qunitjs.com/
[7] https://github.com/airportyh/testem
[8] 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**[9], a **Node.js** library. **Express** not only serves files but also extends Ember CLI 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.

Out of the box, 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. We can also use other testing frameworks like Mocha thanks to the `ember-cli-mocha` addon.

Ember CLI allows us to write our application using **ES6 Modules**. The code is then transpiled (compiled)[10] to **AMD**[11] and finally loaded with the minimalist **AMD** loader, **loader.js**.

We can use **CoffeeScript** if we want, but it is recommended to use plain JS and ES6 modules where possible. In subsequent chapters, we'll explore its syntax and features. Ember CLI ships with Babel[12] support, which allows us to use next generation JavaScript without extra work.

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.

[9]http://expressjs.com/

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

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

[12]Babel

Conventions

We will explore some of the basic conventions and best practices both in `Ember` and `Ember CLI`.

- 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` offers an Ember.Mixin class[13] that you can then reuse in different parts. If you think other people can benefit from this, create an add-on.
- 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`.

[13]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/[14].
2. Install the ember-inspector. Click here for Chrome[15] or here for Firefox[16].
3. Install watchman[17] 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[18]. 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 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.

[14]http://nodejs.org/
[15]https://chrome.google.com/webstore/detail/ember-inspector/bmdblncegkenkacieihfhpjfppoconhi?hl=en
[16]https://addons.mozilla.org/en-US/firefox/addon/ember-inspector/
[17]https://github.com/facebook/watchman
[18]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

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

Change directory into your project.

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: models, routes, templates, components 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` message.

Hands-on

In the following sections we will add models to our application, define the interactions between them, and create an interface to add friends, articles and allow us to keep track of the things we loaned them.

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
installing model
  create app/models/friend.js
installing model-test
  create tests/unit/models/friend-test.js
installing route
  create app/routes/friends.js
  create app/templates/friends.hbs
updating router
  add route friends
installing route-test
  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/[19].

Now let's look at the model and route.

app/models/friend.js

```
// We import the default value from ember-data/model into the variable Model.
//
import Model from 'ember-data/model';
import attr from 'ember-data/attr';

// Define the default export for this model, which will be a subclass
// of ember data 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 Model.extend({

  // attr is the standard way to define attributes with ember data
  firstName: attr('string'),
```

[19] http://jsmodules.io

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


  // ember data expects the attribute **email** on the friend's payload
  email: attr('string'),

  twitter: attr('string')
});
```

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/route'** or **import { Model } from 'ember-data/model'**. This is currently possible in **ES6** using Named Imports and Exports[20].

The following RFC https://github.com/emberjs/rfcs/pull/68[21] offers more context on the move to ES6 modules.

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.

[20]http://jsmodules.io
[21]https://github.com/emberjs/rfcs/pull/68

Connecting with a Backend

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

Verb	URI Pattern	
GET	/friends/:friend_id/relationships/loans(.:format)	
POST	/friends/:friend_id/relationships/loans(.:format)	
PUT	PATCH	/friends/:friend_id/relationships/loans(.:format)
DELETE	/friends/:friend_id/relationships/loans(.:format)	
GET	/friends/:friend_id/loans(.:format)	
GET	/friends(.:format)	
POST	/friends(.:format)	
GET	/friends/:id(.:format)	
PATCH	/friends/:id(.:format)	
PUT	/friends/:id(.:format)	
DELETE	/friends/:id(.:format)	
GET	/articles/:article_id/relationships/loans(.:format)	
POST	/articles/:article_id/relationships/loans(.:format)	
PUT	PATCH	/articles/:article_id/relationships/loans(.:format)
DELETE	/articles/:article_id/relationships/loans(.:format)	
GET	/articles/:article_id/loans(.:format)	
GET	/articles(.:format)	
POST	/articles(.:format)	
GET	/articles/:id(.:format)	
PATCH	/articles/:id(.:format)	
PUT	/articles/:id(.:format)	
DELETE	/articles/:id(.:format)	
GET	/loans/:lend_id/relationships/article(.:format)	
PUT	PATCH	/loans/:lend_id/relationships/article(.:format)
DELETE	/loans/:lend_id/relationships/article(.:format)	
GET	/loans/:lend_id/article(.:format)	
GET	/loans/:lend_id/relationships/friend(.:format)	

Verb	URI Pattern	
PUT	PATCH	/loans/:lend_-id/relationships/friend(.:format)
DELETE	/loans/:lend_-id/relationships/friend(.:format)	
GET	/loans/:lend_id/friend(.:format)	
GET	/loans(.:format)	
POST	/loans(.:format)	
GET	/loans/:id(.:format)	
PATCH	/loans/:id(.:format)	
PUT	/loans/:id(.:format)	
DELETE	/loans/:id(.:format)	

If we do a **GET** request to **http://api.ember-101.com/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-101.com/friends | python -m json.tool
{
  "data": [
    {
      "id": "1",
      "type": "friends",
      "links": {
        "self": "http://api.ember-101.com/friends/1"
      },
      "attributes": {
        "first-name": "Cyril",
        "last-name": "Neveu",
        "email": "cyryl@neveu.com",
        "twitter": null
      },
      "relationships": {
        "loans": {
          "links": {
            "self": "http://api.ember-101.com/friends/1/relationships/loans",
            "related": "http://api.ember-101.com/friends/1/loans"
          }
        }
      }
    }
  ],
  "links": {
    "first": "http://api.ember-101.com/friends?page%5Blimit%5D=10&page%5Boffset%\
5D=0",
    "last": "http://api.ember-101.com/friends?page%5Blimit%5D=10&page%5Boffset%5\
D=0"
  }
}
```

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. Optionally, we can also use JQ https://stedolan.github.io/jq/.

The previous payload follows `JSON API` and returns a list of resource objects which will be used 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[22].

A word on Adapters

Ember data has two mechanism to translate request to the server and transform incoming or outgoing data, such mechanisms are called an adapter and serializer. By default, ember data uses the **DS.JSONAPIAdapter**[23], which expects your API to follow http://jsonapi.org/[24], a specification for building APIs in JSON. In our example, we'll be using this adapter since our API is written using JSON API.

If you want to learn about JSON API while building the application used as example here, then check out the book JSON API By Example[25].

Ember data doesn't force us to use this adapter, we can work with others or create our own. One of those adapters is the active model adapter, built for people using or following API similars to the ones created with active model serializer[26], which uses a different convention for keys and naming. Everything is in **snake_case** and objects are linked in different ways.

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

Some of them are:

[22]https://github.com/abuiles/borrowers-api

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

[24]http://jsonapi.org/

[25]https://leanpub.com/json-api-by-example

[26](https://github.com/rails-api/active_model_serializers)

- ember-data-django-rest-adapter[27]
- ember-data-tastypie-adapter[28]
- emberfire: FireBase adapter[29]

We can find a longer list of adapters if we search GitHub for ember-data adapters[30].

Playing with the resolver

Before going deep into our app, let's talk about something which will be very useful while we build Ember applications and that thing is the resolver.

The resolver is the system in charge of returning whatever Ember requires at different stages, so if we need to load a template, route or service it all goes through the resolver. Normally people don't need to interact with it directly when building applications, but knowing that it is there and being able to idenfity whatever it is trying to do might probably come handy while debugging our applications.

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`. It should look like:

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;
  }
```

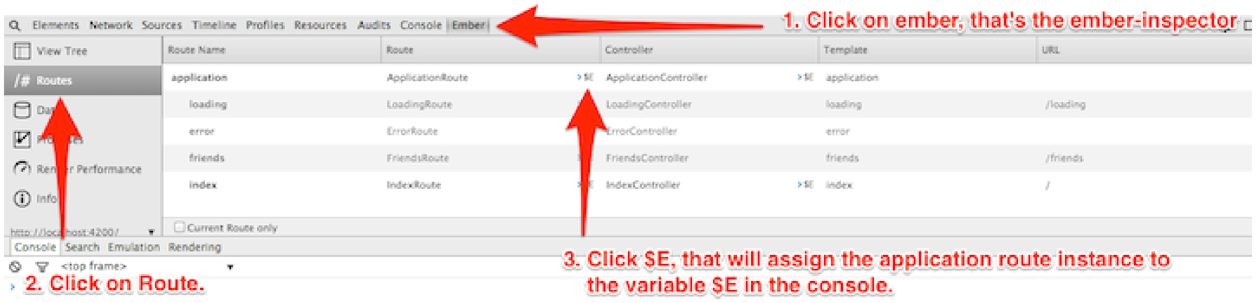
That line will log whatever **ember** tries to "find" to the browser's console. If we go to http://localhost:4200[31] and open the console, we'll see something like the output below:

[27]https://github.com/toranb/ember-data-django-rest-adapter
[28]https://github.com/escalant3/ember-data-tastypie-adapter
[29]https://github.com/firebase/emberfire
[30]https://github.com/search?q=ember-data+adapter&ref=opensearch
[31]http://localhost:4200

```
[ ] 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


Elements Network Sources Timeline Profiles Resources Audits Console Ember
1. Click on ember, that's the ember-inspector
View Tree
/# Routes
Render Performance
Info
http://localhost:4200/
Console Search Emulation Rendering
<top frame>
2. Click on Route.
3. Click $E, that will assign the application route instance to the variable $E in the console.

Route Name	Route	Controller	Template	URL
application	ApplicationRoute $E	ApplicationController $E	application	
loading	LoadingRoute	LoadingController	loading	/loading
error	ErrorRoute	ErrorController	error	
friends	FriendsRoute	FriendsController	friends	/friends
index	IndexRoute $E	IndexController $E	index	/

Current Route only

ember-inspector

We can grab almost any instance of a Route, Controller, Component 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 **application route** 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')
[ ] model:friend   .............borrowers/friend/model
[ ] model:friend   .............borrowers/models/friend
[✓] model:friend   .............borrowers/models/friend
[✓] model:friend   .............borrowers/models/friend
[✓] model:friend   .............borrowers/models/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.

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 **JSONAPIAdapter**. We can check the implementation for this directly in the adapterFor[32] function in **ember data**.

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

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. At the time of this writing a new structure has been propossed for Ember projects and will become the default, more information can be found in the following RFC https://github.com/emberjs/rfcs/pull/143

Once ember data has resolved the adapter it tries to follow the logic to fetch all objects for a given resource. In ours we'll find an error like the following.

```
GET http://localhost:4200/friends 404 (Not Found)
```

The requests failed because we aren't connected to any backend.

We need to 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-101.com**. To do so we use the option **–proxy**:

Running ember server

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

Let's go back to the console and load all our friends, but this time logging 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/friends".
Hi from Cyril
```

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

When calling the store method, we used 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 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-101.com/friends[33]. 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[34] or Chrome Version[35].

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[36] 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.route('friends');**.

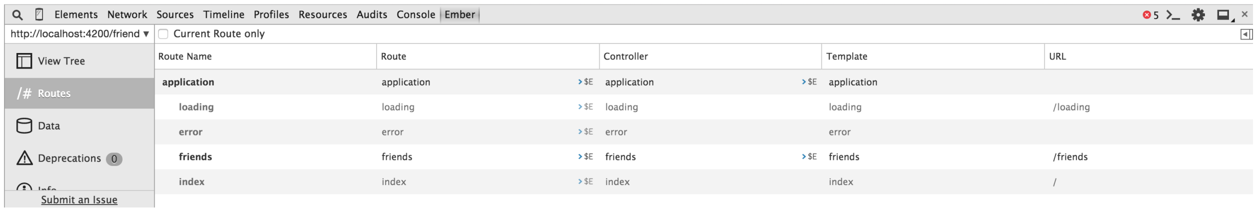
app/router.js

```
// ...

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

// ...
```

We specify the **URLs** we want in our application inside the function passed to **Router.map**. There, we can declare new routes calling **this.route**.

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

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



Route Name	Route	Controller	Template	URL
application	application	application	application	
loading	loading	loading	loading	/loading
error	error	error	error	
friends	friends	friends	friends	/friends
index	index	index	index	/

ember-inspector

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

- ApplicationRoute
- IndexRoute
- LoadingRoute
- ErrorRoute

We also see that the friends route was added with **this.route('friends')**. if we pass a function as second or third argument, **ember** will create an **Index**, **Loading**, and **Error Route**.

We can modify our route passing an empty function

app/router.js

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

And then if we check the inspector, the children routes are automatically generated for the friends route.

When we define a route leaving out the empty function, then the children routes are not generated.

Since we just added a friends index route, visiting http://localhost:4200/friends[37] 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:

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

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**:

app/templates/friends/index.hbs

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

<ul>
  {{#each model as |friend|}}
    <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:

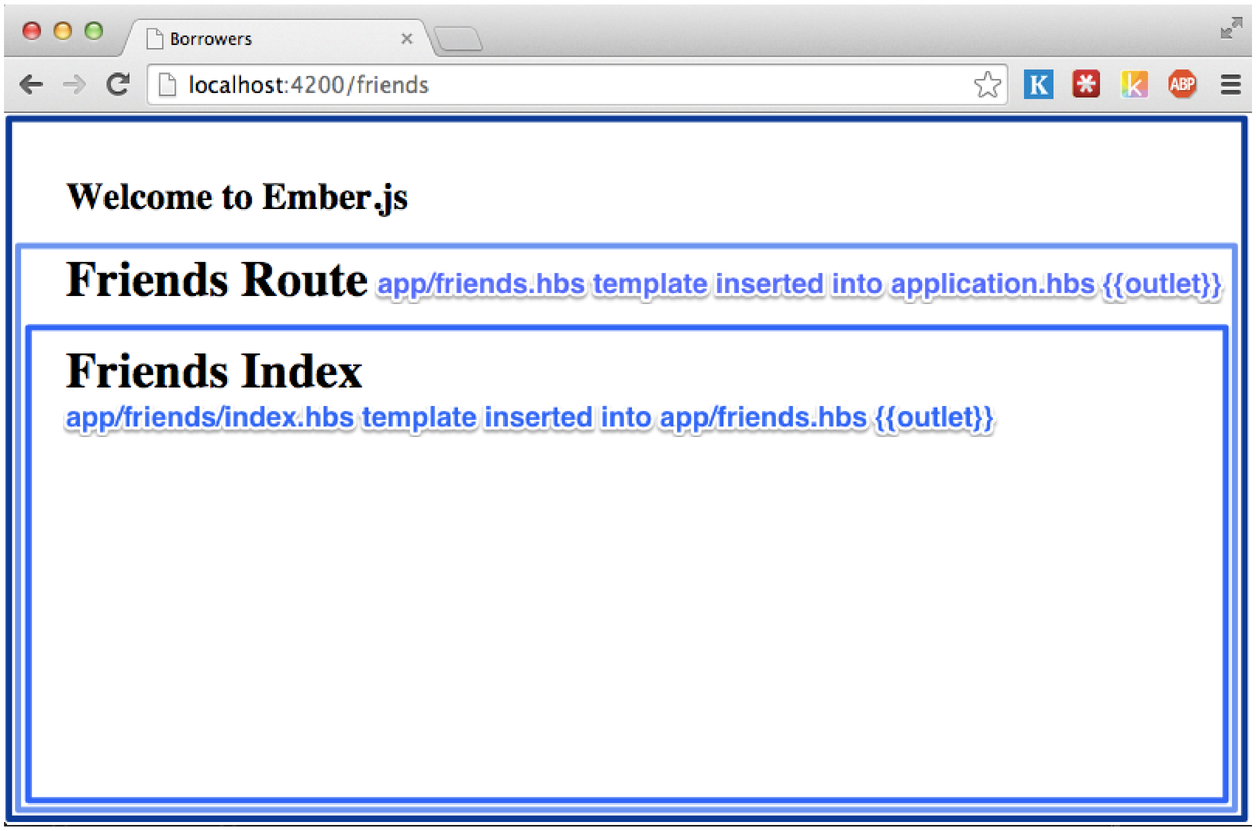
app/routes/friends/index.js

```
import Ember from 'ember';

export default Ember.Route.extend({
  //
  // Here we are using ES6 syntax for functions!
  // We can use this out of the box with ember-cli
  // thanks to the addon ember-cli-babel
  //
  // To learn more about ES6, check http://s.abuiles.com/bwWo
  //
  model() {
    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[38] we will see something like the following along with a list of our friends:

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


Borrowers
localhost:4200/friends
Welcome to Ember.js
Friends Route app/friends.hbs template inserted into application.hbs {{outlet}}
Friends Index
app/friends/index.hbs template inserted into app/friends.hbs {{outlet}}

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**.

Controller will be deprecated in next versions of ember so we'll explore how to work without them in upcoming chapters. We'll still need to use them for some things, so let's not worry if we hear that controllers are disappearing.

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

When we do **{{#each model as |friend|}}**, ember 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 model as |friend|}}
    <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 list our friends, 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 route 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, let's run the route generator with the parameters **friends/new**:

```
$ ember g route friends/new
installing route
  create app/routes/friends/new.js
  create app/templates/friends/new.hbs
updating router
  add route friends/new
installing route-test
  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 route **friends**:

app/router.js

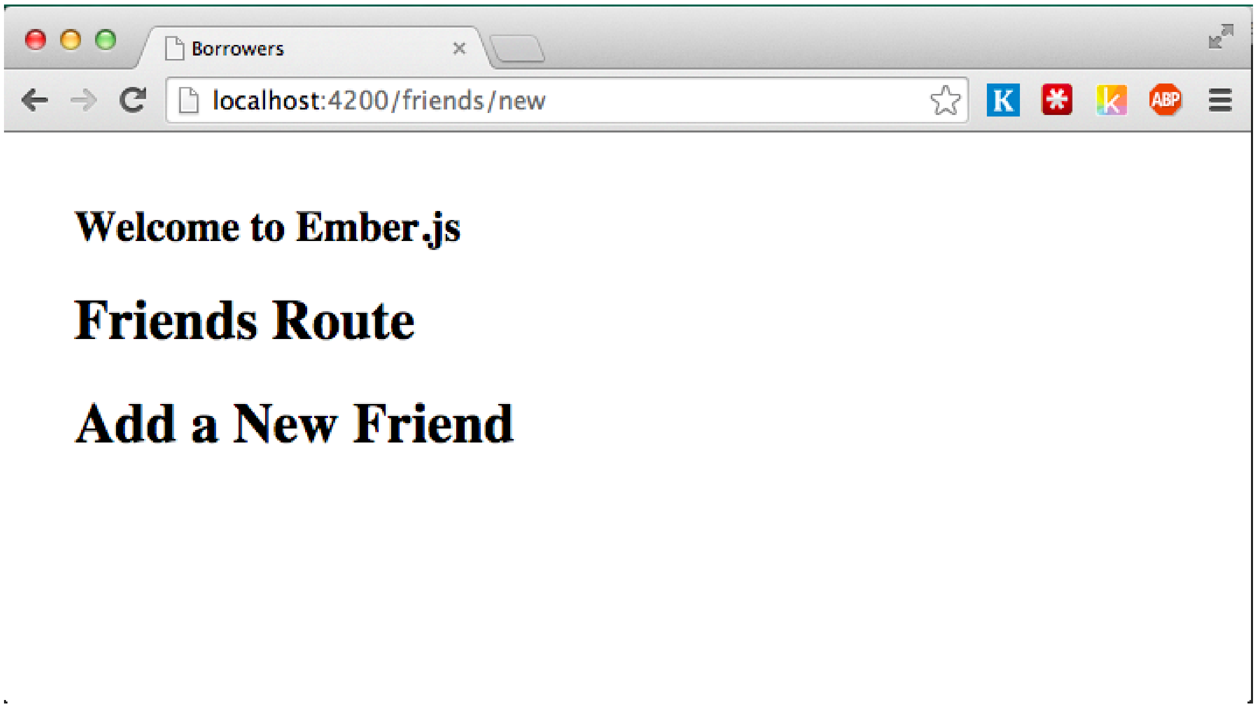
```
this.route('friends', function(){
  this.route('new');
});
```

Let’s 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:



Welcome to Ember.js

Friends Route

Add a New Friend

friends new route

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**, and then 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**.

We need to edit **app/routes/friends/new.js** and add the following model hook:

app/routes/friends/new.js

```
import Ember from 'ember';

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

We have been using the **this.store** without knowing what it is. The Store[39] 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[40].

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 component[41] to contain the form, we can generate the component with the generator, `ember g component friends/edit-form` and add the following content:

app/templates/components/friends/edit-form.hbs

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

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

[40]https://github.com/emberjs/data/blob/v1.13.5/packages/ember-data/lib/initializers/store-injections.js

[41]http://emberjs.com/api/classes/Ember.Component.html

```
      {{input value=model.lastName }}
    </label>
  </p>
  <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>
```

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

app/templates/friends/new.hbs

```
<h1>Adding New Friend</h1>
{{friends/edit-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.

Components

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

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

```
{{friends/edit-form model=model}}
```

This is how components are rendered, we'll have a whole section to talk about components, but for now let's say that they are isolated "templates", they don't know anything about the context surrounding them, so we need to pass down all the necessary data for it to display correctly. In our example, the component required a property called "model" to work, so we are assigning our "current context" model to the component's model.

The friend form is a perfect candidate for a component since we will be using the same form to create and edit a new friend. The only difference will be how "save" and "cancel" will behave under both scenarios.

{{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 **component**, **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 **component**, **controller** or **route**. When the action is called, ember looks for the definition in the current context, so if we are inside the component, it will look at the component.

Let's go to the component and add the actions **save** and **cancel**.

app/components/friends/edit-form.js

```
import Ember from 'ember';

export default Ember.Component.extend({
  actions: {
    save() {
      console.log('+- save action in edit-form component');
    },
    cancel() {
      console.log('+- cancel action in edit-form component');
    }
  }
});
```

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

This action is running on the context of the component so if we do `this.get('model')` we'll get the record created on the model's hook because we passed it down as an argument when rendering the component.

A component not only receives objects but we can also pass it actions, by default the actions need to be specified in the context where we are calling it, and to do so we use the action helper too.

Let's edit our `friends/new` template to add the save and cancel action, it should look like the following now:

app/templates/friends/new.hbs

```
{{friends/edit-form
  model=model
  save=(action "save")
  cancel=(action "cancel")
}}
```

After adding the actions, we'll see the following error in the console:

```
Uncaught Error: An action named 'save' was not found in (generated friends.new c\
ontroller).
```

The issue here is that we didn't specify an action in the `friends/new` controller. As mentioned previously controllers will be replaced eventually but for now if we want to connect a component with its surrounding context then we need to use controllers too.

We can create a controller using the controller generator like `ember g controller friends/new` and then let's add the `save` and `cancel` actions:

app/controllers/friends/new

```
import Ember from 'ember';

export default Ember.Controller.extend({
  actions: {
    save(model) {
      console.log('+--- save action called in friends new controller');
    },
    cancel() {
      console.log('+--- cancel action called in friends new controller');
    }
  }
});
```

Now the route renders again but we won't see the actions in the controller being called yet, the reason is that we need to call the passed action from the component's action. To do so, let's change our component as follows:

app/components/friends/edit-form.js

```
import Ember from 'ember';

export default Ember.Component.extend({
  actions: {
    save() {
      console.log('+- save action in edit-form component');

      //
      // We are calling the save action passed down when rendering the
      // component: action=(action "save")
      //
      this.save(this.get('model'));
    },
    cancel() {
      console.log('+- cancel action in edit-form component');

      //
      // We are calling the cancel action passed down when rendering the
      // component: action=(action "cancel")
      //
      this.cancel();
    }
  }
});
```

How is this related to creating a new friend in our API? We'll discover that after we cover the next helper. On the **save** action in the component, we'll validate our model, and if it is valid call the action in the controller which will take care of calling **.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 an html input field to a property in our model. With the following **{{input value=firstName}}**, changing the value changes the property **firstName**.

Let's modify our component's template to include the following before the form:

[42]http://emberjs.com/api/classes/Ember.Templates.helpers.html#method_input

app/templates/components/friends/edit-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'**.

```
{{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, **{{component}}**, 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 the action `save` which will call **.save()** on the model. Otherwise, we'll see an error message on the form.

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

app/templates/components/friends/edit-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/components/friends/edit-form.js** by adding a computed property called **isValid**:

[43]http://emberjs.com/api/classes/Ember.Checkbox.html

app/components/friends/edit-form.js

```
export default Ember.Component.extend({
  isValid: Ember.computed(
    'model.email',
    'model.firstName',
    'model.lastName',
    'model.twitter',
    {
      get() {
        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:

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

actions in app/components/friends/edit-form.js

```
actions: {
  save() {
    console.log('+- save action in edit-form component');
    if (this.get('isValid')) {
      this.get('model').save().then((friend) => {
        //
        // This function gets called if the HTTP request succeeds
        //
        //
        // We are calling the save action passed down when rendering
        // the component: action=(action "save")
        //
        return this.save(friend);
      }, (err) => {
        //
        // This gets called if the HTTP request fails.
        //
        this.set('errorMessage', 'there was something wrong saving the model');
      });
    } else {
      this.set('errorMessage', 'You have to fill all the fields');
    }
  },
  cancel() {
    console.log('+- cancel action in edit-form component');

    //
    // We are calling the cancel action passed down when rendering the
    // component: action=(action "cancel")
    //
    this.cancel();
  }
}
```

When the action **save** is called, we are first checking if **isValid** is true, then we get the model and call **.save()**. The return of **save()** is a promise, which allows 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 call the save action specified in the controller, in this function we'll put the logic to transiton to the route **friends show** where we can see our friend's profile.

If we click save and have filled all the required fields, we'll see that nothing happens after saving a new friend, but the action `save` was called in the controller. Let's change the controller as follows to include the transition logic:

app/controllers/friends/new

```
import Ember from 'ember';

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

      this.transitionToRoute('friends.show', model);
    },
    cancel() {
      console.log('+--- cancel action called in friends new controller');
    }
  }
});
```

If we save a friend once more, we'll 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, look at the following talk from Ember NYC called The Promise Land[45].

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**

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

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

Route Generator

When creating a new route we can define a custom path for the route with the option `--path`. 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**.

app/router.js

```
this.route('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.findRecord('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 model as |friend|}}
  <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="ember592" href="/friends/1" class="ember-view">
  Cyril Neveu
</a>
```

If our friend model had a property called **fullName**, we could have written the helper with this other form of `link-to` which doesn't include the block:

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 Model from 'ember-data/model';
import attr from 'ember-data/attr';
import { belongsTo } from 'ember-data/relationships';
import Ember from 'ember';

export default Model.extend({
  firstName: attr('string'),
  lastName: attr('string'),
  email: attr('string'),
  twitter: attr('string'),
  fullName: Ember.computed('firstName', 'lastName', {
    get() {
      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** without the block 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. Update the template.
3. Add Controller and actions.

To create the **friends edit route** we should run:

```
$ ember g route friends/edit --path=:friend_id/edit
installing route
  create app/routes/friends/edit.js
  create app/templates/friends/edit.hbs
updating router
  add route friends/edit
installing route-test
  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.route('friends', function() {
    this.route('new');

    this.route('show', {
      path: ':friend_id'
    });

    this.route('edit', {
      path: ':friend_id/edit'
    });
  })
```

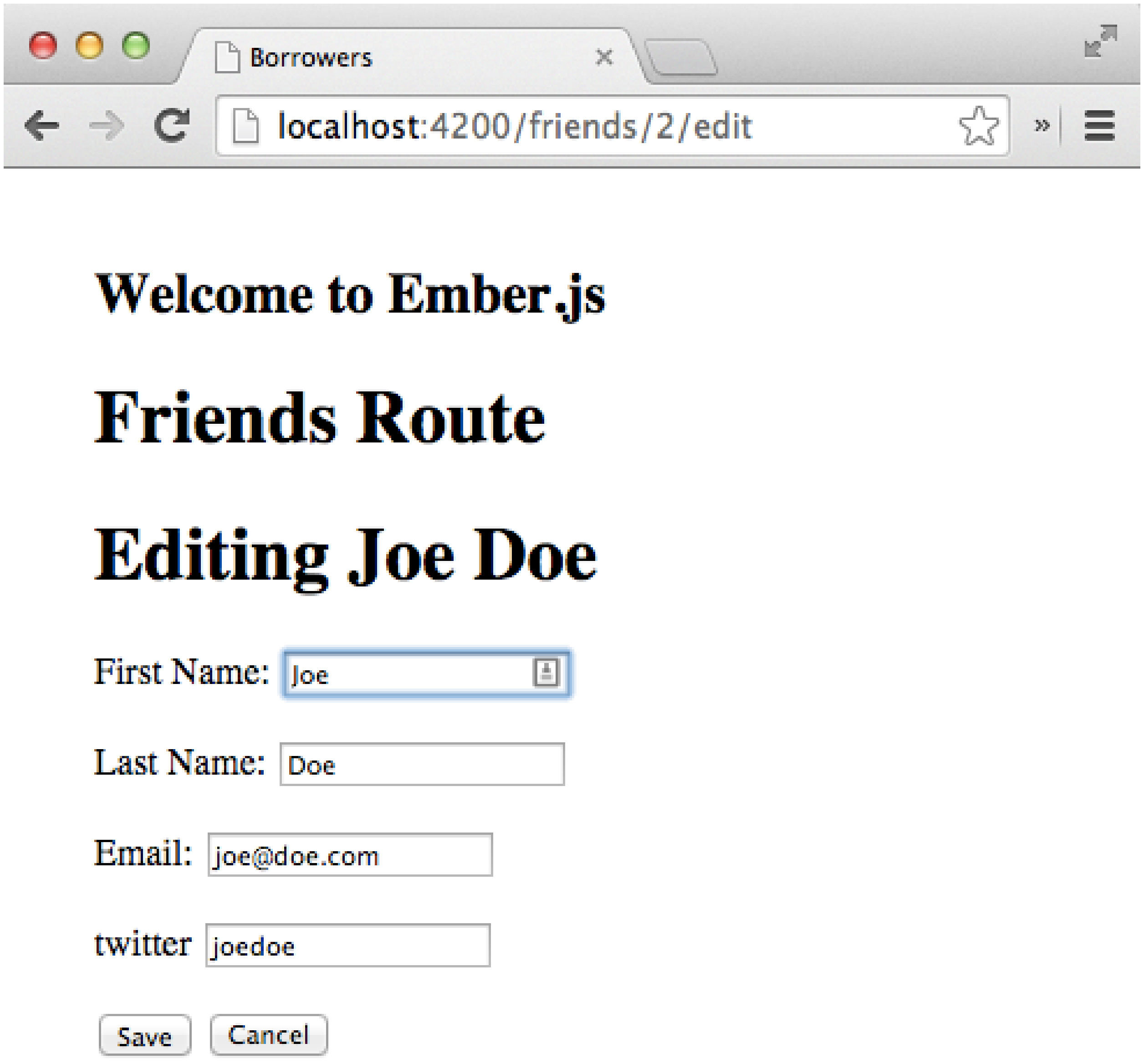
Since the route's path follows the pattern **model_name_id**, we don't need to specify a model hook.

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

app/templates/friends/edit.hbs

```
<h1>Editing {{model.fullName}}</h1>
{{friends/edit-form
  model=model
}}
```

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.)



Welcome to Ember.js

Friends Route

Editing Joe Doe

First Name: Joe

Last Name: Doe

Email: joe@doe.com

twitter joedoe

Friends Edit

Thanks to the component, 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 error because we didn't pass down the `save` and `cancel` actions which our component depends on.

Let's create a `friends.edit` controller and implement those actions. The **save** action will behave exactly as the one in **new**. The action **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
installing controller
  create app/controllers/friends/edit.js
installing controller-test
  create tests/unit/controllers/friends/edit-test.js
```

And then we can add the save and cancel actions:

app/controllers/friends/edit.js

```
import Ember from 'ember';

export default Ember.Controller.extend({
  actions: {
    save(model) {
      this.transitionToRoute('friends.show', model);
    },
    cancel(model) {
      this.transitionToRoute('friends.show', model);
    }
  }
});
```

After adding the actions, if we go to the edit template and click cancel or save we'll still see an error. The problem is that we didn't specify the actions when rendering the component. Let's change the edit form to pass the actions `save` and `cancel`.

app/templates/friends/edit.hbs

```
<h1>Editing {{model.fullName}}</h1>
{{friends/edit-form
  model=model
  save=(action "save")
  cancel=(action "cancel")
}}
```

If we refresh our browser, edit the profile, and click save, we'll see our changes applied successfully. But if we click **cancel** it won't work as expected. The reason is that `cancel` need to receive the model as parameter.

We can fix it by going to the `friends/edit-form` component and call cancel with the model.

app/components/friends/edit-form.js

```
import Ember from 'ember';

export default Ember.Component.extend({
  // ...
  actions: {
    // ...
    cancel() {
      this.cancel(this.get('model'));
    }
  }
});
```

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**.

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.

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()[46] 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 model as |friend|}}
      <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* to the controller.

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

app/controllers/friends/index.js

```
import Ember from 'ember';

export default Ember.Controller.extend({
  actions: {
    delete(friend) {
      friend.destroyRecord();
    }
  }
});
```

To support deletion on **friends show route**, we just need to add the same link with the action delete and implement the action in the controller.

app/controllers/friends/show.js

```
import Ember from 'ember';

export default Ember.Route.extend({
  actions: {
    delete(friend) {
      friend.destroyRecord().then(() => {
        this.transitionToRoute('friends.index');
      });
    }
  }
});
```

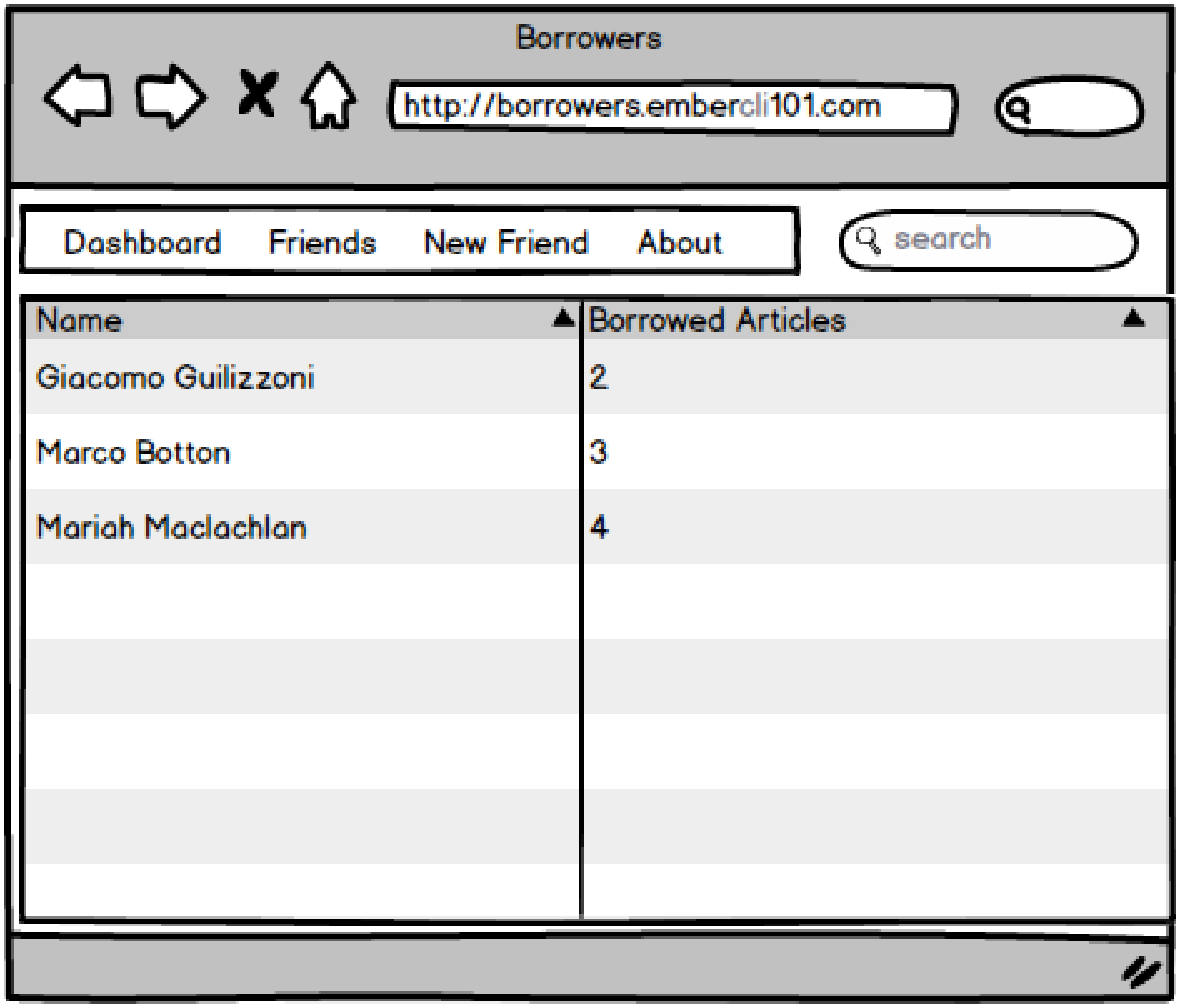
With that we can now create, update, edit, and delete any of our friends!

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



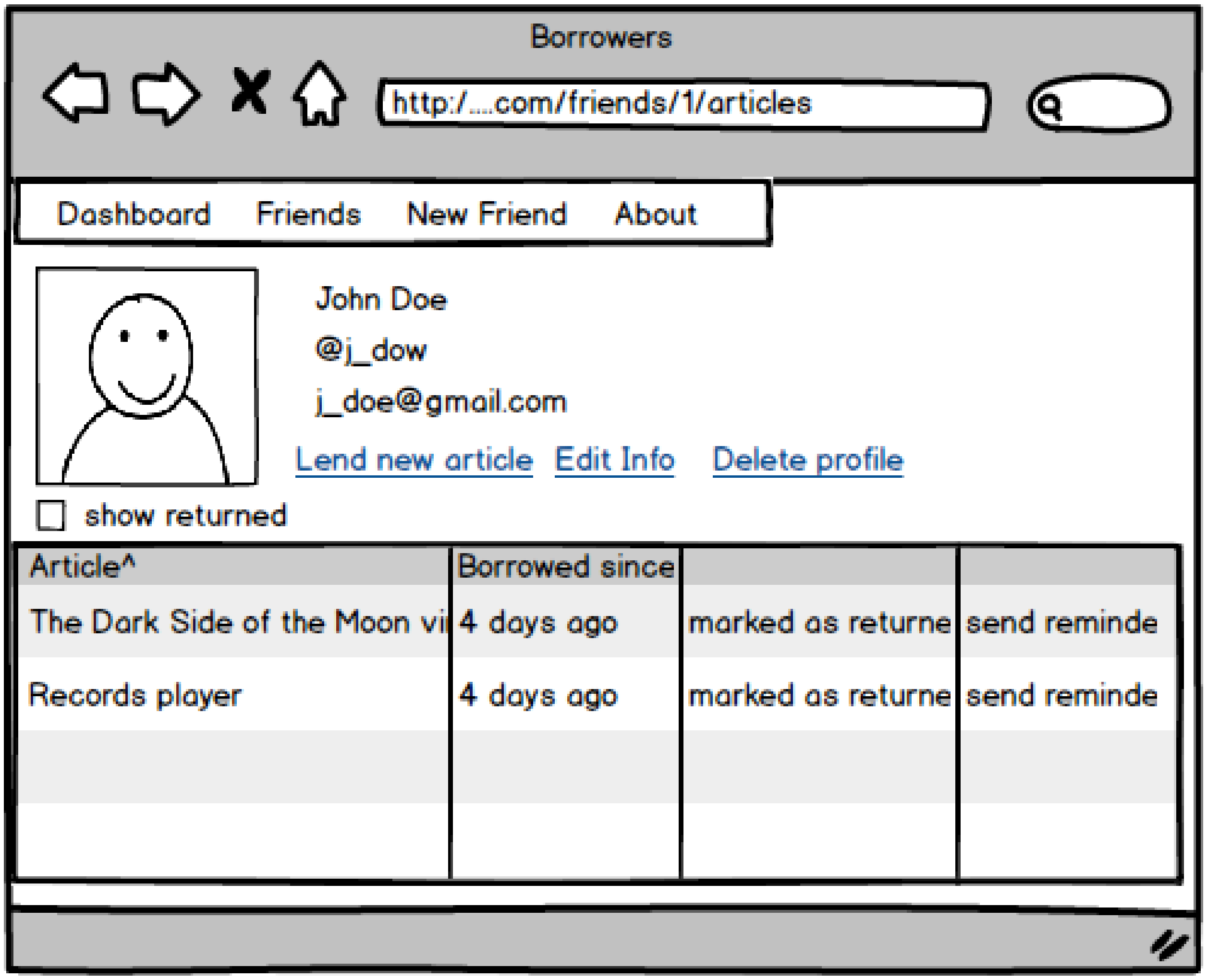
Name ▲	Borrowed Articles ▲
Giacomo Guilizzoni	2
Marco Botton	3
Mariah Maclachlan	4

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



Article^	Borrowed since		
The Dark Side of the Moon vi	4 days ago	marked as returne	send reminde
Records player	4 days ago	marked as returne	send reminde

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/articlesloans**). 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**.

Installing Dependencies

To save time, we'll be using Basscss[47] as our base CSS and **fontello** for icons.

Including Basscss

Basscss is distributed through `npm` or downloading directly from a CDN, but for our example we want to compile it ourselves and put it into our `vendor.css`. To do, let's download the source file and put it in the directory `vendor`.

Adding Basscss to the project

```
curl https://npmcdn.com/basscss@7.1.1/css/basscss.min.css > vendor/basscss.min.c\
ss
```

The fact that a file is in the vendor directory 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 ember-cli-build.js before `return app.toTree();`

Adding Basscss to the ember-cli-build.js

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

module.exports = function(defaults) {
  var app = new EmberApp(defaults, {
  });

  app.import('vendor/basscss.min.css');

  return app.toTree();
};
```

app.import is a helper function that tells **ember CLI** to append **vendor/basscss.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:

[47] http://www.basscss.com/

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** defined under **app/styles**.

Why do we 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 `ember-cli-build.js` we need to stop and start the server again so the changes are applied. Once we have done that, we can refresh our browser and go to **http://localhost:4200/assets/vendor.css**, we'll see that the code for **Basscss** is there.

Including fontello

Because fontello[48] doesn't have a custom distribution either, we can't download it 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 https://www.dropbox.com/s/bo2gi770ydxjc9v/fontello.zip?dl=0 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:

[48]http://fontello.com/

ember-cli-build.js

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

module.exports = function(defaults) {
  var app = new EmberApp(defaults, {
  });

  app.import('vendor/fontello/fontello.css');

  app.import('vendor/fontello/font/fontello.ttf', {
    destDir: 'assets/font'
  });
  app.import('vendor/fontello/font/fontello.eot', {
    destDir: 'assets/font'
  });
  app.import('vendor/fontello/font/fontello.svg', {
    destDir: 'assets/font'
  });
  app.import('vendor/fontello/font/fontello.woff', {
    destDir: 'assets/font'
  });
  app.import('vendor/fontello/font/fontello.woff2', {
    destDir: 'assets/font'
  });

  app.import('vendor/basscss.min.css');

  return app.toTree();
};
```

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 **assets/font**. If we save and refresh our browser, **vendor.css** should now include **fontello.css**.

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 components to simplify our templates. In this case, our component contains the code for the navigation bar. Let's create a component call `nav-bar` and add the following content:

app/templates/components/nav-bar.hbs

```
<header class="h2 border">
  <nav class="flex item-center bg-white">
    {{link-to "Borrowers" "index" class="p2"}}
    {{link-to "Dashboard" "index" class="p2 icon-gauge"}}
    {{link-to "Friends" "friends" class="p2 icon-users-1"}}
    {{link-to "New Friend" "friends.new" class="p2 icon-user-add"}}
  </nav>
</header>
```

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:

app/templates/application.hbs

```
{{nav-bar}}

<main class="clearfix">
  {{outlet}}
</main>
```

We will render the header and wrap the outlet in a row using **basscss** 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="mt3 fit">
  <thead class="p1 h2">
    <tr class="white bg-blue">
      <th>Name</th>
      <th></th>
    </tr>
  </thead>
  <tbody class="p1 h3">
    {{#each model as |friend|}}
      <tr>
        <td class="border-bottom">
          {{link-to friend.fullName "friends.show" friend}}
        </td>
        <td class="border-bottom">
          <a href="#" {{action "delete" friend}}>Delete</a>
        </td>
      </tr>
    {{/each}}
  </tbody>
</table>
```

Now if we visit http://localhost:4200/friends, our friends and navigation bar should look nicer.

New Friend And Friend profile template

Let's edit `app.css` with the following:

app/styles/app.css

```
.borrowers-form .input,
.borrowers-form textarea{
    width: 100%;
    font-size: 1.5em;
    border: 1px solid rgba(196,197,200,.6);
    padding: 30px;
    margin-bottom: 20px;
}
```

And then edit the `friends/edit-form` component:

app/templates/components/friends/edit-form.hbs

```
<form {{action "save" on="submit"}} class="col-8 px2 mx-auto borrowers-form">
  {{#if errorMessage}}
    <h2 class="white bg-red p1">{{errorMessage}}</h2>
  {{/if}}
  {{input value=model.firstName placeholder="First Name" class="input fit"}}<br>
  {{input value=model.lastName  placeholder="Last Name" class="input fit"}}<br>
  {{input value=model.email     placeholder="Email" class="input fit"}}<br>
  {{input value=model.twitter   placeholder="Twitter" class="input fit"}}<br>
  <button {{action "cancel"}} class="btn h3 border white bg-gray p2 mr2 col-3">C\
ancel</button>
  <input type="submit" value="Save" class="btn h3 white bg-green border p2 mr2 c\
ol-3">
</form>
```

And let's center `friends/new` with the following:

app/templates/friends/new

```
<div class="center" >
  <h1>Add a New Friend</h1>
  {{friends/edit-form
  model=model
  save=(action "save")
  cancel=(action "cancel")
  }}
</div>
```

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

app/templates/friends/show.hbs

```
<div class="col-8 px2 mx-auto p1 h2 center">
  <p class="">{{model.firstName}}</p>
  <p class="">{{model.lastName}}</p>
  <p class="">{{model.email}}</p>
  <p class="">{{model.twitter}}</p>
  <p class="">{{link-to "Edit info" "friends.edit" model}}</p>
  <p class=""><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 a simple title:

app/templates/index.hbs

```
**<h2>Dashboard</h2>**.
```

Let's move on with more functionality.

Articles Resource

With our **Friends** CRUD ready, let's create a similar interface to add articles to our system.

Let's create an articles resource:

```
$ ember generate resource articles name: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 Model from 'ember-data/model';
import attr from 'ember-data/attr';

export default Model.extend({
  name: attr('string')
});
```

We have defined our **Articles** model successfully, but we need to create a full interface so we can add or remove articles to our system. We'll add two more tabs to our navigation header, one to list all articles and the other one to add new ones.

app/templates/components/nav-bar.hbs

```
<header class="h2 border">
  <nav class="flex item-center bg-white">
    {{link-to "Borrowers" "index" class="p2"}}
    {{link-to "Dashboard" "index" class="p2 icon-gauge"}}
    {{link-to "Friends" "friends" class="p2 icon-users-1"}}
    {{link-to "New Friend" "friends.new" class="p2 icon-user-add"}}
    {{#link-to "articles" class="p2 icon-motorcycle"}}
      <span></span>Articles
    {{/link-to}}
    {{link-to "New Article" "articles.new" class="p2"}}
  </nav>
</header>
```

After adding this, we'll see an error in the console because we haven't created yet the route for new articles.

Let's create the routes for creating, editing and showing articles. They will be very similar to what we did for friends.

CRUD routes for articles

```
$ ember g route articles/index
$ ember g route articles/new
$ ember g route articles/show --path=:article_id
$ ember g route articles/edit --path=:article_id/edit
```

And then we can wire the functionality following a similar pattern to the one we did for friends. Add a model hook in the index and then render every article. Create a `edit-form` component for articles. Use the `edit-form` in the `new` and `edit` route.

The `articles index` route should look like the following:

app/routes/articles/index.js

```
import Ember from 'ember';

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

And for the template we can reuse the one from friends:

app/templates/articles/index.hbs

```
<table class="mt3 fit">
  <thead class="p1 h2">
    <tr class="white bg-blue">
      <th>Name</th>
      <th></th>
    </tr>
  </thead>
  <tbody class="p1 h3">
    {{#each model as |article|}}
      <tr>
        <td class="border-bottom">
          {{link-to article.name "articles.show" article}}
        </td>
        <td class="border-bottom">
          <a href="#" {{action "delete" article}}>Delete</a>
        </td>
      </tr>
    {{/each}}
  </tbody>
</table>
```

Next we need to create the component for the article form:

```
$ ember g component articles/edit-form
```

And add the following to the component's template:

app/templates/components/articles/edit-form.hbs

```
<form {{action "save" on="submit"}} class="col-8 px2 mx-auto borrowers-form">
  {{#if errorMessage}}
    <h2 class="white bg-red p1">{{errorMessage}}</h2>
  {{/if}}
  {{input value=model.name placeholder="Article name" class="input fit"}}<br>
  <button {{action "cancel"}} class="btn h3 border white bg-gray p2 mr2 col-3">C\
ancel</button>
  <input type="submit" value="Save" class="btn h3 white bg-green border p2 mr2 c\
ol-3">
</form>
```

Next, let's modify the component so it does the validation and saves the model:

app/components/articles/edit-form.js

```
import Ember from 'ember';

export default Ember.Component.extend({
  isValid: Ember.computed.notEmpty('model.name'),
  actions: {
    save() {
      if (this.get('isValid')) {
        this.get('model').save().then((friend) => {
          return this.save(friend);
        }, () => {
          this.set('errorMessage', 'there was something wrong saving the model');
        });
      } else {
        this.set('errorMessage', 'You have to fill all the fields');
      }
    },
    cancel() {
      //
      // We are calling the cancel action passed down when rendering the
      // component: action=(action "cancel")
      //
      this.cancel(this.get('model'));
    }
  }
});
```

Then, we need to change the articles new template to render the component:

app/templates/articles/new.hbs

```
<div class="center" >
  <h1>New Article</h1>
  {{articles/edit-form
    model=model
    save=(action save)
    cancel=(action cancel)
  }}
</div>
```

If we go to http://localhost:4200/articles/new we'll see an error in the conole saying: `An action could not be made for save in <borrowers@controller:articles/new.`

If we look again at the template above, we'll notice that the syntax to call the action is slighty different to the one we use in friends new. Instead of passing the name of the action as a string, we are calling it as if it were a property in the controller.

This is another way of using actions, and they are called closure actions. Introduced in the RFC 0050-improved-actions.md[49], these allow us to pass functions directly as actions. This means we don't need to define the action in the `action` object and we can just bind any function defined in the controller.

To make `save` and `cancel` work, let's create the the articles new controller running `ember g controller articles/new` and then add the following content:

app/controllers/articles/new.js

```
import Ember from 'ember';

export default Ember.Controller.extend({
  save(model) {
    console.log('save action called in articles new');
  },
  cancel() {
    console.log('cancel action called in articles new');
  }
});
```

Once we have defined the actions in the controller then the template should render. Next, if we try to save a new article, we'll see that it doesn't work. The reason is that we didn't create a model hook on the articles new route. Let's do that next:

app/routes/articles/new.js

```
import Ember from 'ember';

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

We will leave as a task the rest of the routes, templates and actions. We still need to support edit, delete and show for articles. Also, use closure actions.

[49]https://github.com/emberjs/rfcs/blob/master/text/0050-improved-actions.md

The commit Add CRUD for articles[50] includes all the changes we did here and the ones left as an exercise.

Defining relationships.

We'll be using a join model to keep track of who borrowed what. The name for this join model will be "loan". A loan belongs to one friend and one article. Friends and articles can have many loans.

In other frameworks this is known as **hasMany** and **belongsTo** relationships, and so are they 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:

```
loans: hasMany('loans')
```

Or we want a **belongsTo**:

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

Let's run the resource generator to create the loan model:

```
$ ember g resource loans notes:string returned:boolean  createdAt:date friend:be\
longsTo article:belongsTo
```

If we open the loan model, it will look something like the following:

[50] https://github.com/abuiles/borrowers-2016/commit/02d8aa23501c9f8be6d636af94ed30a7d366c371

app/models/loan.js

```
import Model from 'ember-data/model';
import attr from 'ember-data/attr';

//
// We can export hasMany or belongsTo depending on the type of the
// relationship.
//
import { belongsTo } from 'ember-data/relationships';

export default Model.extend({
  notes: attr('string'),
  returned: attr('boolean'),
  createdAt: attr('date'),
  friend: belongsTo('friend'),
  article: belongsTo('article'),
});
```

Next, using the relationship types, we can modify our **Article** model:

app/models/article.js

```
import Model from 'ember-data/model';
import attr from 'ember-data/attr';
import { hasMany } from 'ember-data/relationships';

export default Model.extend({
  name: attr('string'),
  loans: hasMany('loan')
});
```

And our **Friend** model:

app/models/friend.js

```
import Model from 'ember-data/model';
import attr from 'ember-data/attr';
import { hasMany } from 'ember-data/relationships';
import Ember from 'ember';

export default Model.extend({
  firstName: attr('string'),
  lastName: attr('string'),
  email: attr('string'),
  twitter: attr('string'),
  loans: hasMany('loan'),
  fullName: Ember.computed('firstName', 'lastName', {
    get() {
      return this.get('firstName') + ' ' + this.get('lastName');
    }
  })
});
```

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

Nested Loans Index

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

To handle this scenario. We need to make sure that **loans** 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

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

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

Router.map(function() {
  this.route('friends', function() {
    this.route('new');

    // Here we are nesting loans under friends/show.
    this.route('show', {
      path: ':friend_id'
    }, function() {
      this.route('loans', {resetNamespace: true}, function() {
      });
    });

    this.route('edit', {
      path: ':friend_id/edit'
    });
  });
});

export default Router;
```

What is resetNamespace?

When nesting routes, ember by default combines the parent routes to form the final route name. In the example above if we had excluded `resetNamespace: true` then the final name for the articles routes would have been `friends/show/loans` instead of `loans`. Also the resolver would have expected us to define our route files inside the `friends show route` directory instead of the top level `articles`. This is a common pattern to use when working with nested resources.

Now let's open the **ember-inspector** and check our newly defined 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="clearfix flex p3 h2">
  <div class="">
    <img src="http://www.fillmurray.com/200/200">
    <p class="">{{model.fullName}}</p>
    <p class="">{{model.email}}</p>
    <p class="">{{model.twitter}}</p>
    <p class="">{{link-to "Edit info" "friends.edit" model}}</p>
    <p class=""><a href="#" {{action "delete" model}}>Delete</a></p>
  </div>
  <div class="flex-auto ml1 loans-container">
    {{outlet}}
  </div>
</div>
```

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

Rendering the index.

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

app/templates/loans/index.hbs

```
<h2>Loans Index</h2>
```

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

app/templates/friends/index.hbs

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

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

If we open the **ember-inspector** and filter by *Current Route only**, we'll see `loans.index` at the last route.

Routes are resolved from top to bottom, so when we navigate to **/friends/1/loans** it will go first to the **application route**, then move to **friends show route** to fetch our friend. Once it is loaded, it will move to **loans index route**.

Next we need to define the model hook for the **loans index route**.

Fetching our friend loans.

Let's add the **loans index route** to the generator and reply 'no' when it asks us if we want to overwrite the template.

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

[?] Overwrite app/templates/articles/index.hbs? No, skip
  create app/routes/articles/index.js
  skip app/templates/articles/index.hbs
updating router
  add route articles/index
installing route-test
  create tests/unit/routes/articles/index-test.js
```

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

app/routes/loans/index.js

```
import Ember from 'ember';

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

In the model hook, we are using a new function this.modelFor[51] that helps us grab the model for any parent route. In this scenario, parent routes are all the ones appearing on top of **articles index route** in the **ember-inspector**.

Once we get the model for **friends show route**, we simply ask for its loans. And that's what we are returning.

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

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

app/templates/loans/index.hbs

```
<table>
  <thead>
    <tr>
      <th>Description</th>
      <th>Borrowed since</th>
      <th></th>
      <th></th>
    </tr>
  </thead>
  <tbody>
    {{#each model as |loan|}}
      <tr>
        <td>{{loan.article.name}}</td>
        <td>{{loan.notes}}</td>
        <td>{{loan.createdAt}}</td>
        <td></td>
        <td></td>
      </tr>
    {{/each}}
  </tbody>
</table>
```

If our friend hasn't borrowed any article yet, we can use the **ember-inspector** to create a loan manually.

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

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('loans').createRecord({notes: 'foo'})
$E.get('loans').createRecord({notes: 'bar'})
```

We will see that our loans list updates automatically with the created records.

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

```
$E.get('loans').createRecord({notes: 'test'}).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.
```

And if we look at the response in the network tab, we'll see that there is an error with the message `article - can't be blank`. Let's grab an article and then create a loan including the article.

```
$E.get('store').findAll('article')
article = $E.get('store').peekAll('article').get('firstObject')
$E.get('loans').createRecord({notes: 'a loan', article: article}).save()
```

In the previous snippet, we are using two methods from ember data's store, the first makes a request to download all the available articles in the API and puts them in the store, and the second one is "fetching" all the records from the store (without firing a HTTP request) and we call `firstObject` with returns the first record in the list.

Then, with that record we are creating a new loan.

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

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 **loans**:

app/router.js

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

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

Router.map(function() {
  this.route('friends', function() {
    this.route('new');

    this.route('show', {
      path: ':friend_id'
    }, function() {
      this.route(loans, {resetNamespace: true}, function() {
        this.route('new');
      });
    });

    this.route('edit', {
      path: ':friend_id/edit'
    });
  });

  // ...
});

export default Router;
```

Then let's create the route **app/routes/loans/new.js** with the model hook:

app/routes/loans/new.js

```
import Ember from 'ember';

export default Ember.Route.extend({
  model() {
    return this.store.createRecord('loan', {
      friend: this.modelFor('friends/show')
    });
  }
});
```

In the model hook we use this.store.createRecord[52], 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**, which will make sure that the loan is linked with our friend. For the article we still need to add something like a select where the user can choose an article.

Ember data allows us to specify a **defaultValue** for our attributes. We can use that to set notes as an empty string. In **app/models/loan.js**, let's replace the definition of **notes** so it looks as follows:

app/models/loan.js

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

Next we need to add the **new** template. Since we might want to reuse the **form**, let's add it as a component and then include it in the template.

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

We can follow a similar pattern to the one used with articles and friends. Let's create a component called **loans/edit-form** and then pass the necessary data for it to function properly.

Let's run the following command: `ember g component loans/edit-form` and then edit the template with the form.

app/templates/components/loans/edit-form.hbs

```
<form {{action "save" on="submit"}} class="">
  {{#if errorMessage}}
    <h2 class="white bg-red p1">{{errorMessage}}</h2>
  {{/if}}
  <label>Select an article</label>
  <select class="select">
    <option>Article 1</option>
    <option>Article 2</option>
    <option>Article 3</option>
  </select>
  <br>
  {{textarea value=model.notes placeholder="Notes" class="textarea" cols="50" ro\
ws="10"}}
  <br>
  <button {{action "cancel"}} class="btn border white bg-gray">Cancel</button>
  <input type="submit" value="Save" class="btn white bg-green border">
</form>
```

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

app/templates/loans/new.hbs

```
{{loans/edit-form model=model class="ml3"}}
```

We almost have the basics ready. 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="clearfix flex p3 h2">
  <div class="">
    <img src="http://www.fillmurray.com/200/200">
    <p class="">{{model.fullName}}</p>
    <p class="">{{model.email}}</p>
    <p class="">{{model.twitter}}</p>
    <p>{{link-to "Lend article" "loans.new"}}</p>
    <p class="">{{link-to "Edit info" "friends.edit" model}}</p>
    <p class=""><a href="#" {{action "delete" model}}>Delete</a></p>
  </div>
  <div class="flex-auto ml1 loans-container">
    {{outlet}}
  </div>
</div>
```

We are creating the link with `{{link-to "Lend article" "loans.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 article" "loans.new" friend}}** where **friend** is an instance of a **friend**.

Completing the form

We have the route, template and component for new loans but the select is not showing the list of articles that we can loan to a friend. Let's build that next, and then connect the `cancel` and `save` actions.

Instead of building the select ourselves, we are going to use an addon called ember-power-select[53] which give us a nice select component.

Ember addons offer us an easy way to share code and augment ember-cli, we'll be building one in a subsequent chapter, but for now, let's start consuming them.

We can install the addon running the ember install command, we need to stop the ember-cli server and then run the following:

```
ember install ember-power-select
```

Once our addon has been installed, we can edit the `loans/edit-form` to use it:

[53]http://www.ember-power-select.com/

app/templates/components/loans/edit-form.hbs

```
<form {{action "save" on="submit"}} class="borrowers-form">
  {{#if errorMessage}}
    <h2 class="white bg-red p1">{{errorMessage}}</h2>
  {{/if}}
  <label>Select an article</label>
  {{#power-select class="select"
     selected=model.article
     options=articles
     onchange=(action (mut model.article)) as |article|}}
    {{article.name}}
  {{/power-select}}
  <br>
  {{textarea value=model.notes
    placeholder="Notes"
    class="textarea"
  }}
  <br>
  <button {{action "cancel"}} class="btn border white bg-gray">Cancel</button>
  <input type="submit" value="Save" class="btn white bg-green border">
</form>
```

As we can see, the addon takes as options the current selected option, which in our example would be `model.article`. Then it takes the list of options which will be used to populate the select and also has an action which gets called every time the selection changes.

In the action we are using the `mut` helper which give us a special action to update a property. The following `onchange=(action (mut model.article))` can be read as “On change, change the property `model.article` which the selected valued”. We are not seeing the parameter, but the addon passes it explicitly.

We could have also written a function in the component like the following:

```
changeArticle(article) {
  this.set('model.article', article)
}
```

And then pass it as the `onchange` action: `onchange=(action changeArticle)`.

If we start the server and go to the form for new loans, we’ll see that the select is being displayed but the articles are not listed, the reason for that is that we haven’t populated the articles attribute.

To do so, we’ll be creating a computed property in the component and then calling `store.findAll`, let’s go to `loans/edit-form` component an add the following:

app/components/loans/edit-form.js

```
import Ember from 'ember';

export default Ember.Component.extend({
  //
  // By default the store is not injected into components, so we use
  // the "inject.service" helper to make it available.
  //
  store: Ember.inject.service(),
  articles: Ember.computed({
    get() {
      //
      // Since we are using Ember.inject.service, we need to call the
      // store using the get helper
      //
      return this.get('store').findAll('article');
    }
  }).readOnly()
});
```

Now if we refresh again, we'll see that the list of articles are now in the select.

Next, we need to bind the `new` and `cancel` actions.

After successfully making a new loan or clicking cancel, we want to navigate back to the `loans.index` for the friend. To do so, let's write an action in the controller that we can use in both scenarios.

Let's create the controller using the generator: `ember g controller loans/new` and then add the following content:

app/controllers/loans/new.js

```
import Ember from 'ember';

export default Ember.Controller.extend({
  backToIndex(friend) {
    this.transitionToRoute('loans.index', friend);
  }
});
```

Then in the `loans.new` template, pass down the action:

app/templates/loans/new.hbs

```
{{loans/edit-form model=model back=(action backToIndex) class="ml3"}}
```

And finally we need to connect the `back` action in the component and change the way we call the `save` and `cancel` actions:

app/components/loans/edit-form.js

```
import Ember from 'ember';

export default Ember.Component.extend({
  //
  // By default the store is not injected into components, so we use
  // the "inject.service" helper to make it available.
  //
  store: Ember.inject.service(),
  articles: Ember.computed({
    get() {
      //
      // Since we are using Ember.inject.service, we need to call the
      // store using the get helper
      //
      return this.get('store').findAll('article');
    }
  }).readOnly(),
  //
  // Save and cancel are not declared inside actions key, we'll be
  // using it as closure actions.
  //
  save() {
    //
    // We probably want to verify here that the model has an article
    // before saving
    //
    this.get('model').save().then((model) => {
      this.back(model.get('friend'));
    }, () => {
      this.set(
        'errorMessage',
        'there was something wrong saving the loan'
      );
    });
  },
  cancel() {
    this.back(this.get('model.friend'));
  }
});
```

To finish, we need to change the `save` and `cancel` actions so we use the closure action format instead of “quoted” format.

app/templates/components/loans/edit-form.hbs

```
<form {{action save on="submit"}} class="borrowers-form">
  {{#if errorMessage}}
    <h2 class="white bg-red p1">{{errorMessage}}</h2>
  {{/if}}
  <label>Select an article</label>
  {{#power-select class="select"
     selected=model.article
     options=articles
     onchange=(action (mut model.article)) as |article|}}
    {{article.name}}
  {{/power-select}}
  <br>
  {{textarea value=model.notes
    placeholder="Notes"
    class="textarea"
  }}
  <br>
  <button {{action cancel}} class="btn border white bg-gray">Cancel</button>
  <input type="submit" value="Save" class="btn white bg-green border">
</form>
```

If we go to a friend profile and click “Lend article”, we’ll be able to create a new loan.

Computed Property Macros

In **app/components/friends/edit-form.js**, we define the computed property **isValid** with the following code:

Computed Property isValid in app/components/friends/edit-form.js

```
isValid: Ember.computed(
  'model.email',
  'model.firstName',
  'model.lastName',
  'model.twitter',
  {
    get() {
      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[54]
- Ember.computed.notEmpty[55]

[54]http://emberjs.com/api/classes/Ember.computed.html#method_and
[55]http://emberjs.com/api/classes/Ember.computed.html#method_notEmpty

Computed Property With Macros in app/components/friends/edit-form.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 than original implementation.

We can see the full list of computed properties under the Ember.computed namespace[56].

Using components to mark a loan as returned.

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

We'll add a select in the loans index, so we can mark an item as returned or borrowed. Whenever that loan has pending changes, we'll see a **save** button.

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

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

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

Creating an component

```
ember g component loans/loan-row
```

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

[56]http://emberjs.com/api/classes/Ember.computed.html

app/components/loans/loan-row.js

```
import  Ember from 'ember';

export default Ember.Component.extend({
  tagName: 'tr',
  loan: null // passed-in
});
```

We are specifying that the html tag 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 `loan` and `loanStates` with value `null` and the comment: "passed-in". It will help people consuming the component to identify which data they should pass-in.

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

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

```
<td>{{loan.article.name}}</td>
<td>{{loan.notes}}</td>
<td>{{loan.createdAt}}</td>
<td>
  {{input type="checkbox" checked=loan.returned}}
</td>
<td>
  {{#if loan.isSaving}}
    <p>Saving ...</p>
  {{else if loan.hasDirtyAttributes}}
    <button {{action save loan}}>Save</button>
  {{/if}}
</td>
```

In the template we are defining the cells for every loan row and reading the value from the "passed-in" property loan, also we are calling the action `save`, which we are passing in.

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

The previous properties are part of DS.Model[57] and they help us to know things about a model. In the previous scenario, **loan.hasDirtyAttributes** becomes true if there is a change to the model and **loan.isSaving** is true if the model tries to persist any changes to the backend.

Before using our components, let's create the `loans.index` controller and add a function called `save` which we'll use to save changes in a loan.

Let's add the following content after running `ember g controller loans/index`:

[57]http://emberjs.com/api/data/classes/DS.Model.html

app/controllers/loans/index.js

```
import Ember from 'ember';

export default Ember.Controller.extend({
  save(loan) {
    return loan.save();
  }
});
```

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

app/templates/loans/index.hbs

```
<table>
  <thead>
    <tr>
      <th>Article</th>
      <th>Notes</th>
      <th>Borrowed since</th>
      <th></th>
      <th></th>
    </tr>
  </thead>
  <tbody>
    {{#each model as |loan|}}
      {{loans/loan-row
        loan=loan
        save=(action save)
      }}
    {{/each}}
  </tbody>
</table>
```

We are iterating over every loan in the model and then rendering an loans-row component for each of them, we are passing as attributes the loan, bounding the save action to another action which is also called save.

In upcoming versions of ember, we'll be able to use components as if they were just another HTML tag, so we could write <loans/loan-row> instead of {{loans/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[58]: * isDeleted * hasDirtyAttributes * isEmpty * isError * isLoaded * isLoading * isNew * isReloading * isSaving * isValid

If we go to the browser and try what we just created, everything should work.

Implementing auto save.

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

We'll rewrite our template so the save button is not included and then use the input helper action[59] to call save.

app/templates/components/loans/loan-row.hbs

```
<td>{{loan.article.name}}</td>
<td>{{loan.notes}}</td>
<td>{{loan.createdAt}}</td>
<td>
  {{input type="checkbox" checked=loan.returned click=(action save loan)}}
</td>
<td>
  {{#if loan.isSaving}}
    <p>Saving ...</p>
  {{/if}}
</td>
```

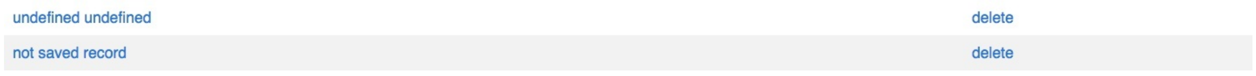
We are doing something new here, we are passing the argument to the action in the action itself like `action save loan`, this is one of the cool things we can do with closure actions.

Now, every time we change the state of the loan, the `save` action will be called.

Route hooks

If we go to http://localhost:4200/friends/new[60] 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.

[58]http://emberjs.com/api/data/classes/DS.Model.html#property_isDeleted
[59]https://guides.emberjs.com/v2.5.0/templates/input-helpers/#toc_actions
[60]http://localhost:4200/friends/new


undefined undefined delete
not saved record delete

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.

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**:

```
model() {
  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 need to remove the record from the store. How do we do that?

Ember.Route[61] has a set of hooks that are called at different times during the route lifetime. For instance, we can use activate[62] to do something when we enter a route, deactivate[63] when we leave it or resetController[64] to reset values on some actions.

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

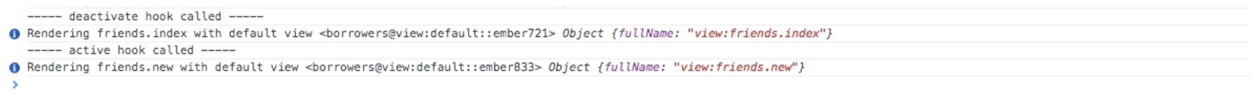
[61]http://emberjs.com/api/classes/Ember.Route.html
[62]http://emberjs.com/api/classes/Ember.Route.html#method_activate
[63]http://emberjs.com/api/classes/Ember.Route.html#method_deactivate
[64]http://emberjs.com/api/classes/Ember.Route.html#method_resetController

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

```
import Ember from 'ember';

export default Ember.Route.extend({
  model() {
    return this.store.createRecord('friend');
  },
  activate() {
    console.log('----- activate hook called -----');
  },
  deactivate() {
    console.log('----- deactivate hook called -----');
  },
  resetController: function (controller, isExiting, transition) {
    if (isExiting) {
      console.log('----- resetController hook called -----');
    }
  }
});
```

And then visit http://localhost:4200/friends/new[65] and click cancel or friends.

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 **resetController** hook to clean up our state.

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

[65]http://localhost:4200/friends/new

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

```
import Ember from 'ember';

export default Ember.Route.extend({
  model() {
    return this.store.createRecord('friend');
  },
  resetController(controller, isExiting) {
    if (isExiting) {
      // We grab the model from the controller
      //
      var model = controller.get('model');

      // Because 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 resetController 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 show whatever change we leave unsaved. To solve this problem we'll use the **resetController** 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 `hasDirtyAttributes`.

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

```
import Ember from 'ember';

export default Ember.Route.extend({
  resetController(controller, isExiting) {
    if (isExiting) {
      var model = controller.get('model');
      model.rollback();
    }
  }
});
```

Tasks

We have the same problem on the routes, **articles.index**, *articles.new** and **loans.new**. Implement the **resetController** hook so unsaved articles and loans 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 **format-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 **format-date** using Momentjs[66], 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 **ember-cli-build.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": "∼2.13.0"** (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[67]

Let's add the following to our **ember-cli-build.js**:

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

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

Next, if we navigate to http://localhost:4200[68], 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:

[68] 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[69] 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:

[69] 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: format-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 **format-date** with the command **ember g helper format-date**, and then modify **app/helpers/format-date.js** so it consumes our format function.

Format date helper app/helpers/format-date.js

```
import Ember from 'ember';

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

export default Ember.Helper.helper(function([date, format]) {
  return formatDate(date, format);
});
```

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

Using format-date in app/components/loans/loan-row.hbs

```
<td>{{loan.article.name}}</td>
<td>{{loan.notes}}</td>
<td>{{format-date loan.createdAt "LL"}}</td>
<td>
  {{input type="checkbox" checked=loan.returned click=(action save loan)}}
</td>
<td>
  {{#if loan.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.

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[70], an "Ember-friendly **jQuery.ajax** wrapper." If we navigate to the lib/main.js[71], we'll notice that the source of the application is written with **ES6** syntax, but it is distributed[72] 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.

Let's add the library to bower with `bower install ic-ajax --save`. If we are asked to select a version of ember, then we need to select the one which is higher. Once it's installed, let's import it into our **ember-cli-build.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.

[70] https://github.com/instructure/ic-ajax/tree/v2.0.1/lib
[71] https://github.com/instructure/ic-ajax/blob/master/lib/main.js
[72] https://github.com/instructure/ic-ajax/tree/v2.0.1/dist

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()  {
    return request('/friends').then(function(data){
      return {
        friendsCount: data.data.length
      };
    });
  }
});
```

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

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

Next, 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/[73].

ember-ajax

We installed `ic-ajax` only for demonstration purposes but this library is not longer the recommended way to wrap ajax request in our ember applications. Instead, there is now a library called ember-ajax[74] which is the one officially supported by the community.

The library `ember-ajax` is consumed a bit differently than `ic-ajax` since it requires us to use a service.

Now that we understand how importing named AMD libraries works, we can remove the **import** for **ic-ajax** from the **ember-cli-build.js** and use `ember-ajax`. Let's run the following commands and then stop and start the server.

[73]http://localhost:4200/
[74]https://github.com/ember-cli/ember-ajax

```
$ bower uninstall ic-ajax --save
$ ember install ember-ajax
```

If we navigate to the dashboard, we'll see an error like `Uncaught Error: Could not find module` ic-ajax `imported from` borrowers/routes/index`. We need to change the route to consume` ember-ajax` instead, it should look like the following:

app/routes/index.js

```
import Ember from 'ember';

export default Ember.Route.extend({
  //
  // Check the following for more information about services
  // https://guides.emberjs.com/v2.5.0/applications/services/
  //
  ajax: Ember.inject.service(),
  model()  {
    return this.get('ajax').request('/friends').then(function(data){
      return {
        friendsCount: data.data.length
      };
    });
  }
});
```

After changing the route with the code above, our application should work fine again.

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[75].

The following is the content of the library:

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

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 **ember-cli-build.js**:

Consuming borrowers-dates

```
app.import('bower_components/borrowers-dates/index.js');
```

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

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

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


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

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[76] 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[77] 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[78].

Using ember-browserify

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

```
$ ember install 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 remove moment with `bower uninstall moment --save` and also remove `app.import('bower_components/moment/moment.js');` from `ember-cli-build.js`.

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`.

[76] http://browserify.org/
[77] https://github.com/ef4
[78] https://github.com/ef4/ember-browserify

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
installing addon
  create .bowerrc
  create .editorconfig
  create .ember-cli
  create .jshintrc
  create .travis.yml
```

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

```
.
|-- LICENSE.md
|-- README.md
|-- addon
|-- app
|-- bower.json
|-- bower_components
|-- config
|-- ember-cli-build.js
|-- 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
installing component
  create addon/components/fill-murray.js
  create addon/templates/components/fill-murray.hbs
installing component-test
  create tests/integration/components/fill-murray-test.js
installing component-addon
  create app/components/fill-murray.js
```

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

addon/components/fill-murray.js

```
import Ember from 'ember';
import layout from '../templates/components/fill-murray';

export default Ember.Component.extend({
  layout: layout,
  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', {
    get() {
      let base = 'http://www.fillmurray.com/';
      let url  = `${base}${this.get('width')}/${this.get('height')}`;

      return url;
    }
  })
});
```

Next we need to specify the body of our component in **addon/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",
```

Since our addon ships with templates, we need to include `ember-cli-htmlbars` in the dependencies, which we can do running `npm i ember-cli-htmlbars --save`.

With the previous 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:

```
$ ember install 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=200 height=200}}
```

Let's modify our **app/templates/friends/show.hbs** to look like the following:

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

```
<div class="clearfix flex p3 h2">
  <div class="">
    {{fill-murray width=200 height=200}}
    <p class="">{{model.fullName}}</p>
    <p class="">{{model.email}}</p>
    <p class="">{{model.twitter}}</p>
    <p>{{link-to "Lend article" "loans.new"}}</p>
    <p class="">{{link-to "Edit info" "friends.edit" model}}</p>
    <p class=""><a href="#" {{action "delete" model}}>Delete</a></p>
  </div>
  <div class="flex-auto ml1 loans-container">
    {{outlet}}
  </div>
</div>
```

After editing the template, the Bill Murray placeholder will appear when we visit a friend's profile. With that we have created and published our first addon.

Ember Data

In this chapter we'll cover some of the public methods from the DS.Store[79] 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.

peekAll

store.peekAll 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.peekAll('friend')
friends.get('length')
> 0
friends.mapBy('firstName')
> []
```

We stored the result in a variable called friends, which is a collection with zero elements. This makes sense because we haven't loaded any **friends** yet. If we click on the friends link and run the following:

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

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

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 **peekAll** is a **live array**. That's why our **friends** variable was updated without requiring any additional steps.

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[80].

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.query`, 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.query('friend', {sort: 'first-name'})

XHR finished loading: GET "http://localhost:4200/api/v2/friends?&sort=first-name\
".
```

In the previous request we asked **query** to load all the articles, sending as parameters the key **sort**.

Like **peekAll** 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.

findRecord: Loading a single record

If we want to load a single record then we should use store.findRecord[81] . To do that, we use the name of the model and the record's **id** as second argument:

[80]https://www.igvita.com/slides/2012/devtools-tips-and-tricks/#4
[81]http://emberjs.com/api/data/classes/DS.Store.html#method_findRecord

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

peekRecord

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

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 create 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**.

There are two ways to work with relationships. The first is working with records pre-loaded into the store, and the second is to load them on demand.

As we know, our API follows JSON API, which gives us different strategies to load the records associated with a relationship. In our API, we use the links strategy[82]. Ember data follows those links to fill up the association automatically.

If we inspect the payload for friends, the results look something like the following:

[82]http://jsonapi.org/format/#document-resource-object-relationships

```
{
  "attributes": {
    "created-at": "2016-06-12T13:47:02.427Z",
    "email": "arya@got.com",
    "first-name": "Arya",
    "last-name": "Stark",
    "twitter": "@noone"
  },
  "id": "64",
  "links": {
    "self": "http://api.ember-101.com/friends/64"
  },
  "relationships": {
    "loans": {
      "links": {
        "related": "http://api.ember-101.com/friends/64/loans",
        "self": "http://api.ember-101.com/friends/64/relationships/loans"
      }
    }
  },
  "type": "friends"
}
```

This includes the model's attributes and a key called `relationships` which includes the links where the related attributes can be fetched.

Ember data then follows that link to get the list of loans for a friend.

We can see how this works if we visit `http://localhost:4200/friends` and then click on a friend, we'll see that there are several requests after that one to fetch the related loans and then more requests to fetch the related articles.

We can avoid all those requests sideloading the related records, JSONAPI give us a way to do so, which is using the parameter `include` and then a path for the things we want to include.

Let's modify the index route so it includes the query parameter `include` with the value `loans`:

app/routes/friends/index.js

```
import Ember from 'ember';

export default Ember.Route.extend({
  model() {
    //
    // We now use store.query and pass include in the options
    //

    return this.store.query('friend', {include: 'loans'});
  }
});
```

Now if we go to to index, we'll see that the request for the "loans" relationship is not happening, but there are request for articles.

We can sideload the articles too, but we need to ask for that in the `include` query params. Let's modify our route once more so it looks like the following:

app/routes/friends/index.js

```
import Ember from 'ember';

export default Ember.Route.extend({
  model() {
    //
    // We now use store.query and pass include in the options
    //

    return this.store.query('friend', {include: 'loans,loans.article'});
  }
});
```

If we go once more to friends and then click on one of them, we won't see extra requests happening.

We can see the difference in the payload visiting the following page http://api.ember-101.com/friends?include=loans and then adding or removing things to the query param `include`.

[83]http://api.ember-101.com/friends?include=loans

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', {
    get() {
      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', {
  get() {
    return this.get('firstName') + ' ' + this.get('lastName');
  }
})
```

We can also add support for setting the value of a computed property and handling how it should behave. The following example shows how:

Computed Property with set support

```
fullName: Ember.computed('firstName', 'lastName', {
  get() {
    return this.get('firstName') + ' ' + this.get('lastName')
  },

  set(key, value) {
    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[84].

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 Read-Only.

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[85] wrote an **ember-cli-addon** called ember-computed-change-gate[86] 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:

[84]https://github.com/emberjs/ember.js/blob/v2.6.0/packages/ember-metal/lib/computed.js#L77
[85]https://twitter.com/gavinjoyce
[86]https://github.com/GavinJoyce/ember-computed-change-gate

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 supports 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[87], which allows us to keep track of changes in any property or computed property.

Let's use an observer on the low-row component to do something every time the state changes, let's add the following:

app/components/loans/loan-row.js

```
  stateChanged: Ember.observer('loan.returned', function() {
    var loan = this.get('loan');
    console.log('OMG Expensive operation because loan state changed');
  }),
```

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.

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[88]

[87]http://en.wikipedia.org/wiki/Observer_pattern

[88]https://github.com/emberjs/guides/pull/110

We can also create an observer using `addObserver` from Ember.Observable[89]. We could define the `stateChanged` observer like this:

```
import Ember from 'ember';

export default Ember.Component.extend({
  tagName: 'tr',
  loan: null, // passed-in
  init() {
    this._super(...arguments);
    this.addObserver('loan.returned', this, this.stateChanged);
  },
  stateChanged() {
    console.log('OMG Expensive operation because loan state changed');
  }
});
```

Here we are using `init` which is the first function called when the component is started, and then we call _`super()` to call the function on the object's parent. Calling super is not always required but if you are using mixins or inheriting from a class that does something on `init` then we need to call it.

The ember guides have a great section on the component life cycle, we recommend to give that section a read to expand the knowledge in this area https://guides.emberjs.com/v2.6.0/components/the-component-lifecycle/

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 .[][90], which is just a special handler that changes every time the collection content changes.

The second one is @each[91], which allows us to observe properties on each of the items in the collection.

We can use the previous function in our loans index to call a function when we add a new loan, and then the other one when we change the state of a loan:

89http://emberjs.com/api/classes/Ember.Observable.html
90http://emberjs.com/api/classes/Ember.Array.html#property__
91http://emberjs.com/api/classes/Ember.Array.html#property__each

app/controllers/loans/index.js

```
import Ember from 'ember';

export default Ember.Controller.extend({
  save(loan) {
    return loan.save();
  },
  contentDidChange: Ember.observer('model.[]', function() {
    console.log('Called when we add or removed a loan.');
  }),
  stateDidChange: Ember.observer('model.@each.returned', function() {
    console.log('Called when the state property change for any of the loans.');
  })
});
```

If we visit any of our friends' profiles and change the state for any loan 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[92] gave a talk called Stop Breaking the Web[93].

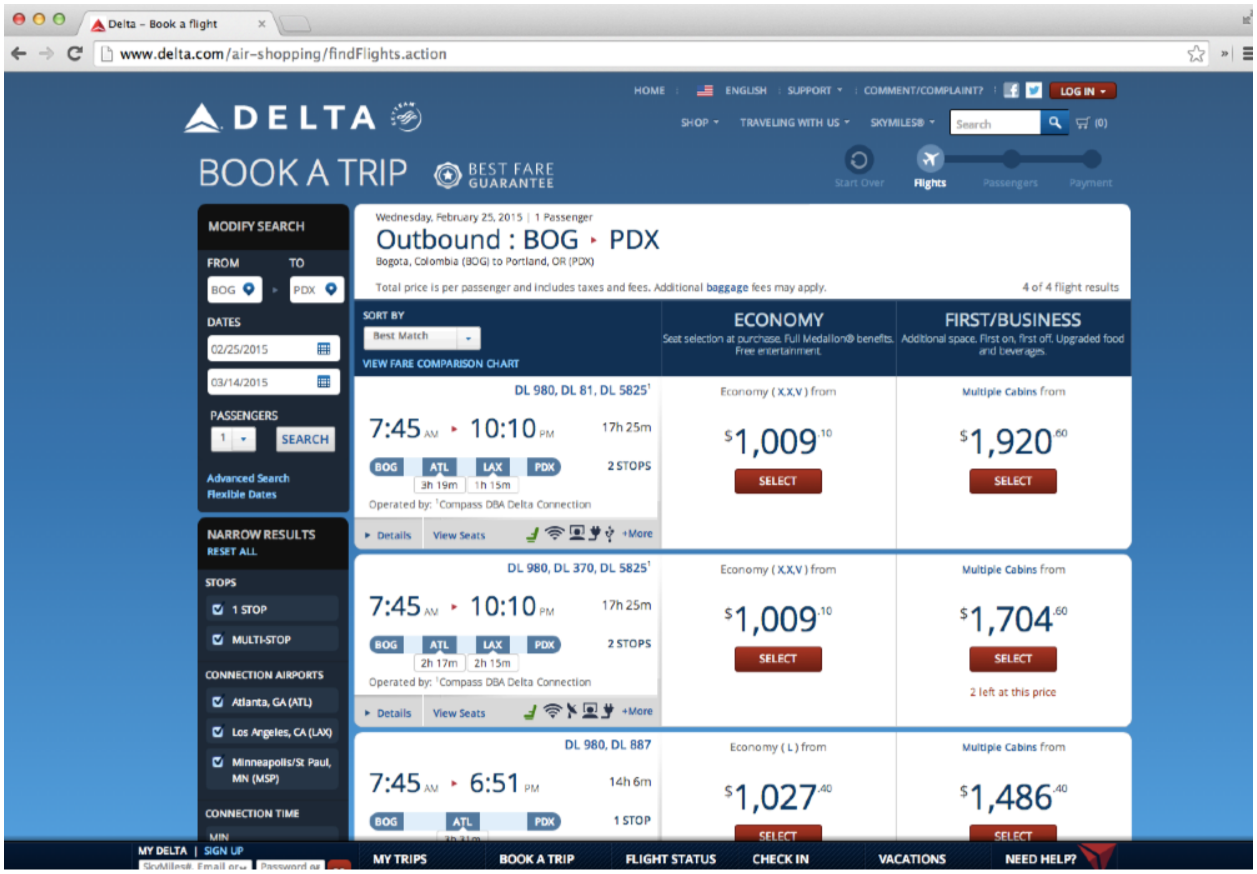
Tom talks about the importance of the URL and how we should give it a higher priority in our applications. Ideally, the URL should be able to reflect our application state in such a way that we can easily reference it, bookmark it, or share it with others.

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.

[92]https://twitter.com/tomdale

[93]http://2013.jsconf.eu/speakers/tom-dale-stop-breaking-the-web.html


Delta – Book a flight
www.delta.com/air-shopping/findFlights.action
HOME ENGLISH SUPPORT COMMENT/COMPLAINT? LOG IN
DELTA
SHOP TRAVELING WITH US SKYMILES® Search (0)
BOOK A TRIP BEST FARE GUARANTEE
Start Over Flights Passengers Payment
MODIFY SEARCH
NARROW RESULTS RESET ALL
SORT BY Best Match
VIEW FARE COMPARISON CHART
ECONOMY
Seat selection at purchase. Full Medallion® benefits. Free entertainment
FIRST/BUSINESS
Additional space. First on, first off. Upgraded food and beverages
SELECT
Details View Seats +More
MY DELTA | SIGN UP
MY TRIPS BOOK A TRIP FLIGHT STATUS CHECK IN VACATIONS NEED HELP?

FROM BOG TO PDX
DATES 02/25/2015 03/14/2015
PASSENGERS 1 SEARCH
Advanced Search
Flexible Dates
STOPS
☑ 1 STOP
☑ MULTI-STOP
CONNECTION AIRPORTS
☑ Atlanta, GA (ATL)
☑ Los Angeles, CA (LAX)
☑ Minneapolis/St Paul, MN (MSP)
CONNECTION TIME

Wednesday, February 25, 2015 | 1 Passenger
Outbound : BOG ▸ PDX
Bogota, Colombia (BOG) to Portland, OR (PDX)
Total price is per passenger and includes taxes and fees. Additional baggage fees may apply. 4 of 4 flight results

DL 980, DL 81, DL 5825
7:45 AM ▸ 10:10 PM 17h 25m
BOG ATL LAX PDX 2 STOPS
3h 19m 1h 15m
Operated by: Compass DBA Delta Connection
Economy (X,X,V) from $1,009.10
Multiple Cabins from $1,920.60

DL 980, DL 370, DL 5825
7:45 AM ▸ 10:10 PM 17h 25m
BOG ATL LAX PDX 2 STOPS
2h 17m 2h 15m
Operated by: Compass DBA Delta Connection
Economy (X,X,V) from $1,009.10
Multiple Cabins from $1,704.60
2 left at this price

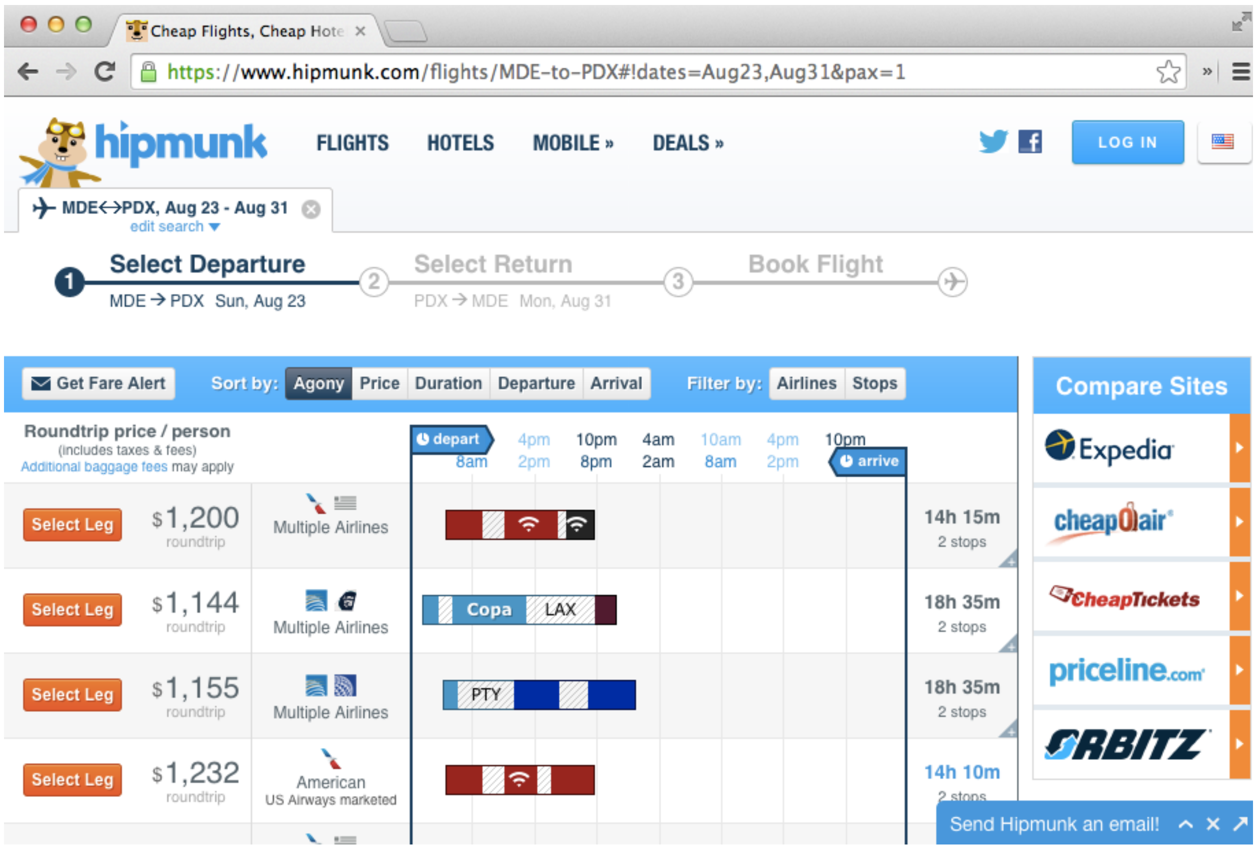
DL 980, DL 887
7:45 AM ▸ 6:51 PM 14h 6m
BOG ATL PDX 1 STOP
Economy (L) from $1,027.40
Multiple Cabins from $1,486.40

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 another browser, we'll get a bunch of errors and not the search we originally performed.

Now let's do a search on hipmunk[94]. This website places greater value on the functionality of the URL.

[94]https://www.hipmunk.com


Cheap Flights, Cheap Hote
https://www.hipmunk.com/flights/MDE-to-PDX#!dates=Aug23,Aug31&pax=1
hipmunk
FLIGHTS
HOTELS
MOBILE »
DEALS »
LOG IN
MDE↔PDX, Aug 23 - Aug 31
edit search
Select Departure
MDE → PDX Sun, Aug 23
Select Return
PDX → MDE Mon, Aug 31
Book Flight
Get Fare Alert
Sort by: Agony Price Duration Departure Arrival
Filter by: Airlines Stops
Compare Sites
Roundtrip price / person
(includes taxes & fees)
Additional baggage fees may apply
Select Leg $1,200 roundtrip Multiple Airlines 14h 15m 2 stops
Select Leg $1,144 roundtrip Multiple Airlines Copa LAX 18h 35m 2 stops
Select Leg $1,155 roundtrip Multiple Airlines PTY 18h 35m 2 stops
Select Leg $1,232 roundtrip American US Airways marketed 14h 10m 2 stops
Expedia
cheapOair
CheapTickets
priceline.com
ORBITZ
Send Hipmunk an email!

hipmunk search

The search above results in the following: flights/MDE-to-PDX#!dates=Aug23,Aug31&pax=1[95]. 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 a function `setSortBy` that will receive as parameter the field we want to sort our properties by.

[95]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: 'first-name',
  //
  // 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;
  },
  actions: {
    delete(friend) {
      friend.destroyRecord();
    }
  }
});
```

Now we need to call the `setSortBy` action in the **app/templates/friends/index.hbs**

app/templates/friends/index.hbs

```
<table class="mt3 fit">
  <thead class="p1 h2">
    <tr class="white bg-blue">
      <th {{action setSortBy "first-name"}}> Name</th>
      <th></th>
    </tr>
  </thead>
  <tbody class="p1 h3">
    {{#each model as |friend|}}
      <tr>
        <td class="border-bottom">
          {{link-to friend.fullName "loans" friend}}
        </td>
        <td class="border-bottom">
          <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 column:

app/styles/app.css

```
[data-ember-action] {
  cursor: pointer;
}
```

Now if we go to http://localhost:4200/friends and click on **Name**, we'll see that our action is being fired and something like the following logged to the browser's console:

```
Sorting by  first-name
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[96] that allows us to persist application state in the URL as parameters, generating URLs like `/friends?sort=first-name`.

[96] 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: 'first-name',
  // ...
```

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 `first-name`.

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 `first-name`, we can write it like this: `{{#link-to 'friends' (query-params sortBy="first-name")}}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 JSONAPI supports `sort`, 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(params) {
    //
    // We use now store.query and pass include in the options
    //

    let query = { include: 'loans,loans.article' };

    // We check if this value is passed
    if (params.sortBy) {
      query.sort = params.sortBy;

      // If sortBy is passed we check if sortAscending is false,
      // and use the JSONAPI syntax for descending sort
      // http://jsonapi.org/format/#fetching-sorting
      //
      if (!params.sortAscending) {
        query.sort = `-${query.sort}`;
      }
    }
    return this.store.query('friend', query);
  }
});
```

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:

```
/friend?sort=first-name
```

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/[97].

[97]https://guides.emberjs.com/v2.6.0/routing/query-params/

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[98], 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.

[98]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[99] 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[100], 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 Ember from 'ember';
import { moduleForModel, test } from 'ember-qunit';

moduleForModel('friend', 'Unit | Model | friend', {
  // Specify the other units that are required for this test.
  needs: ['model:loan']
});

test('it exists', function(assert) {
  let model = this.subject();
  // let store = this.store();
  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');
```

[99] http://localhost:4200/tests
[100] http://phantomjs.org/

```
});

test('loans relationship', function(assert) {
  var klass  = this.subject({}).constructor;

  var relationship = Ember.get(klass, 'relationshipsByName').get('loans');

  assert.equal(relationship.key, 'loans');
  assert.equal(relationship.kind, 'hasMany');
});
```

We can run our tests by going directly to the following URL: http://localhost:4200/tests?module=Friend[101].

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 dateHelpers from 'borrowers/utils/date-helpers';
import { module, test } from 'qunit';

module('Unit | Utility | date helpers');

test('formats a date object', function(assert) {
  var date = new Date("11-3-2014");
  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=Unit%20%7C%20Utility%20%7C%20date%20helpers[10

[101]http://localhost:4200/tests?module=Friend
[102]http://localhost:4200/tests?module=Unit%20%7C%20Utility%20%7C%20date%20helpers

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.

Ember has a set of helpers to simplify writing these kinds of tests. There are synchronous[103] and asynchronous[104] 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 { test } from 'qunit';
import moduleForAcceptance from 'borrowers/tests/helpers/module-for-acceptance';

moduleForAcceptance('Acceptance | friends/new');

test('visiting /friends/new', function(assert) {
  visit('/friends/new');

  andThen(function() {
    assert.equal(currentURL(), '/friends/new');
  });
});
```

Now we can run our tests by visiting http://localhost:4200/tests[105] or, if we want to run only the acceptance tests for Friends New, http://localhost:4200/tests?module=Acceptance%20%7C%20friends%2Fnew[106].

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.

[103]http://emberjs.com/guides/testing/test-helpers/#toc_wait-helpers
[104]http://emberjs.com/guides/testing/test-helpers/#toc_asynchronous-helpers
[105]http://localhost:4200/tests
[106]http://localhost:4200/tests?module=Acceptance%20%7C%20friends%2Fnew

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:

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 the community recommendation is to use ember-cli-mirage[107], a library that allows us to mock servers with a simple DSL.

Let's stop the server and install mirage running `ember install ember-cli-mirage`.

Next, let's disable `mirage` if the environment is development, since we only want to use it for testing.

config/environment.js

```
if (environment === 'development') {
  ENV['ember-cli-mirage'] = {
    enabled: false
  };
}
```

If we go to http://localhost:4200/tests we'll see an error in the acceptance test like:

```
Error: Mirage: Your Ember app tried to GET '/friends',
    but there was no route defined to handle this request.
    Define a route that matches this path in your
    mirage/config.js file. Did you forget to add your namespace?@ 269 ms
```

That's because we didn't add the end-point friends to our mirage server.

Let's tell `mirage` about the routes for friends, adding the following lines to the file `mirage/config.js`:

[107] http://www.ember-cli-mirage.com/

mirage/config.js

```
this.get('/friends');
this.post('/friends');
this.get('/friends/:id');
this.patch('/friends/:id');
this.delete('/friends/:id')
```

After refreshing the tests page, we'll see a new error like the following:

```
Error: Pretender intercepted GET /friends but encountered an error:
Mirage: The route handler for /friends is trying to access the friend
model, but that model doesn't exist. Create it using 'ember g
mirage-model friend'.
```

Mirage is telling us that there is a handler for `friends` but it doesn't know about that model, we need to create the model in mirage with the command `ember g mirage-model friend`.

If we refresh again, the acceptance tests will pass. Mirage also help us if we want to create fake data in the server. To do so, we need to create a factory and then fill the server with fake data before running the test. Let's explore that next.

We can create factories with the `mirage-factory` generator, to create one for friends we can run the following command `ember g mirage-factory friend` and then let's edit the factory to declare the attributes:

```
//
// Mirage ships with Fajer.js which help us to create fake data
//
// See https://github.com/marak/Faker.js/
//
import { Factory, faker } from 'ember-cli-mirage';

export default Factory.extend({
  firstName() {
    return faker.name.firstName();
  },
  lastName() {
    return faker.name.lastName();
  },
  email() {
    return faker.internet.email();
  },
  twitter: '@someone'
});
```

Once we have defined the factory, we can use it in our acceptance test. Let's create one for the friends index route and then assert that the all the friend are rendered.

First, let's create the acceptance test running `ember g acceptance-test friends`

tests/acceptance/friends-test.js

```
import { test } from 'qunit';
import moduleForAcceptance from 'borrowers/tests/helpers/module-for-acceptance';

moduleForAcceptance('Acceptance | friends', {
  beforeEach() {
    //
    // Mirage makes the variable server avaialble in our tests.
    //
    // We can use server.create('model-name') to create 1 entry in the
    // mock server or use createList to create many.
    //
    //
    server.createList('friend', '10');
  }
});

test('visiting /friends', function(assert) {
  visit('/friends');

  andThen(function() {
    assert.equal(currentURL(), '/friends');

    //
    // This will fail since we are creating 10 friends, fix it :)
    //
    assert.equal(
      find('table tbody tr').length,
      9,
      'assertion');

  });
});
```

With this we can have some idea of what mirage allow us to do, we can work with relationships and use mirage for development, but that's out of the scope of this book, for more information check out the documentation http://www.ember-cli-mirage.com/docs/v0.2.x/[108].

Further Reading

During EmberConf 2014, Eric Berry[109] gave a great talk called The Unofficial, Official Ember Testing Guide[110] 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/[111].

To know more about using mocks and fixtures, we recommend the following presentation: Real World Fixtures[112] by Chris Ball[113].

---

[108]http://www.ember-cli-mirage.com/docs/v0.2.x/

[109]https://twitter.com/coderberry

[110]http://www.confreaks.com/videos/3310-emberconf2014-the-unofficial-official-ember-testing-guide

[111]http://emberjs.com/guides/testing

[112]https://speakerdeck.com/cball/real-world-fixtures

[113]https://twitter.com/cball_

PODS

This section has been removed until the Module Unification RFC arrives[114].

For more information about future project structure, we recommend the following talk: Ember.js NYC, May 2016: Robert Jackson on app structure[115]

[114]https://github.com/emberjs/rfcs/pull/143

[115]https://www.youtube.com/watch?v=b3mTGXjm10g&feature=youtu.be&t=22m50s

Deploying Ember.js applications

In this chapter we'll explore different alternatives to deploy our Ember.js applications. We'll talk about S3 and Pagefront based deployments where our application is completely separated from our API.

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[116] 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: 'https://api.ember-101.com',
    // ...
```

Now we can use it in the adapter. Let's create the application adapter running `ember g adapter application` and then add the following:

[116] https://guides.emberjs.com/v2.6.0/models/customizing-adapters/#toc_host-customization

app/adapters/application.js

```
import JSONAPIAdapter from 'ember-data/adapters/json-api';
import config from '../config/environment';

export default JSONAPIAdapter.extend({
  host: config.host
});
```

We also need to change `app/routes/index.js` to use the host:

app/routes/index.js

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

export default Ember.Route.extend({
  //
  // Check the following for more information about services
  // https://guides.emberjs.com/v2.5.0/applications/services/
  //
  ajax: Ember.inject.service(),
  model()  {
    return this.get(`${config.host}/friends`).then(function(data){
      return {
        friendsCount: data.data.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
ember build
Building
Built project successfully. Stored in "dist/".
```

Inspecting the `dist` directory, we'll see the following contents:

```
dist/
|-- assets
|-- |-- app.scss
|-- |-- borrowers.css
|-- |-- borrowers.js
|-- |-- borrowers.map
|-- |-- ember-basic-dropdown.scss
|-- |-- ember-power-select
|-- |-- |-- themes
|-- |-- |-- |-- bootstrap.scss
|-- |-- |-- variables.scss
|-- |-- ember-power-select.scss
|-- |-- failed.png
|-- |-- font
|-- |-- |-- fontello.eot
|-- |-- |-- fontello.svg
|-- |-- |-- fontello.ttf
|-- |-- |-- fontello.woff
|-- |-- |-- fontello.woff2
|-- |-- passed.png
|-- |-- test-loader.js
|-- |-- test-support.css
|-- |-- test-support.js
|-- |-- test-support.map
|-- |-- tests.js
|-- |-- tests.map
|-- |-- vendor.css
|-- |-- vendor.js
|-- |-- vendor.map
|-- crossdomain.xml
|-- 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[117] 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:

```
dist/
|-- assets
|-- |-- app.scss
|-- |-- borrowers-0434066470763b4f6d79622e2bb60ffb.css
|-- |-- borrowers-b01655dbaa5eaca990dcb5f786c3a2b9.js
|-- |-- ember-basic-dropdown.scss
|-- |-- ember-power-select
|-- |-- |-- themes
|-- |-- |-- +-- bootstrap.scss
|-- |-- +-- variables.scss
|-- |-- ember-power-select.scss
|-- |-- font
|-- |-- |-- fontello.eot
|-- |-- |-- fontello.svg
|-- |-- |-- fontello.ttf
|-- |-- |-- fontello.woff
|-- |-- +-- fontello.woff2
|-- |-- vendor-6d05d317665a43d23c85eafd86e90654.css
|-- +-â”€ vendor-8422fb84936d9c467a1a18d2ad95d6c5.js
|-- crossdomain.xml
```

[117] http://localhost:4200/tests

```
|-- index.html
+-â”€ 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-6d05d317665a43d23c85eafd86e90654.css">
<link rel="stylesheet" href="assets/borrowers-0434066470763b4f6d79622e2bb60ffb.c\
ss">
```

Fingerprinting is achieved using broccoli-asset-rev[118]. This allows us the option to select the format of the files we want to fingerprint and to append an URL to every asset.

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[119]

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-101.s3-website-us-east-1.amazonaws.com/[120]

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/*"
                }
        ]
}
```

[118]https://github.com/rickharrison/broccoli-asset-rev
[119]http://docs.aws.amazon.com/AmazonS3/latest/dev/WebsiteHosting.html
[120]http://ember-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[121]

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 for deployments would be to use the ember addon ember-cli-deploy[122]. Their website has a complete guide on how to use their addon.

Deploying to Pagefront

Pagefront[123] is a PaaS for ember. This is probably the easiest way to deploy ember applications today. They have an ember addon which makes the whole process super simple.

First let's install the addon with `ember install`, we'll use `ember-101-GITHUB-HANDLE` to identiy the application:

```
ember install ember-pagefront --app=ember-101-abuiles
```

After installing **ember-pagefront**, a new file `config/deploy.js` will be created with the required data to deploy to pagefront.

Next we can run `ember deploy production` and that's it. Our application will be deployed to https://ember-101-abuiles.pagefrontapp.com[124]

[121]http://docs.aws.amazon.com/gettingstarted/latest/swh/website-hosting-intro.html
[122]http://ember-cli-deploy.com
[123]https://www.pagefronthq.com/
[124]https://ember-101-abuiles.pagefrontapp.com

ember-cli-deploy

Ember CLI deploy[125] has become the default solution to manage deployments with ember. Pagefront use it under the hood, and there are many other plugins available, which allow us to deploy to different platforms or just develop our own solutions.

We can find all the available options visiting the plugins section in their website http://ember-cli-deploy.com/docs/v0.6.x/plugins[126].

[125]http://ember-cli-deploy.com/
[126]http://ember-cli-deploy.com/docs/v0.6.x/plugins

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.

The best way to upgrade is to follow the release notes, which the core team adds with every release.

If we look at the following release https://github.com/ember-cli/ember-cli/releases/tag/v2.7.0-beta.4[127], we'll find notes about the commands that we need to run, notable deprecation's, and links to an example project called ember-new-output[128] which shows the changes after upgrading a project.

Wrapping up.

Ember can have a tough learning curve, but after getting the basics, you'll see that the time invested in learning will pay off. Not only it will make your more productive, but you'll be able to take advantage of its addon ecosystem and all the good stuff happening in the JavaScript world. As Yehuda Katz said during Ember Conf 2016: Eventually all the good ideas will end up in Ember.

Learning Ember today is very different to the time when this book was first published. We have an official "Learning team", they are doing a terrific job keeping the guides up to date[129] and making sure that every new feature is documented for mainstream adoption.

If you are asking yourself were to go next, my recommendation would be start building applications, since that will be the only way to get real world experience. I'll try to update the book as new features arrive and let you know what has changed in the release notes. You can also follow me on twitter abuiles[130] for updates.

Lastly, if you enjoy this book and want to support my work, you can do so buying my other book https://leanpub.com/json-api-by-example[131] or if you download this book for free and think it was worth something, you can send me donations clicking here[132].

Happy learning!

- Adolfo Builes

[127]https://github.com/ember-cli/ember-cli/releases/tag/v2.7.0-beta.4
[128]https://github.com/ember-cli/ember-new-output/compare/v2.7.0-beta.3...v2.7.0-beta.4
[129]https://guides.emberjs.com
[130]https://twitter.com/abuiles
[131]https://leanpub.com/json-api-by-example
[132]https://www.paypal.com/cgi-bin/webscr?cmd=_s-xclick&hosted_button_id=JY94CMEJKR45S