Extracted from:

Deliver Audacious Web Apps with Ember 2

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Deliver Audacious Web Apps with Ember 2

Matthew White

edited by Katharine Dvorak

Deliver Audacious Web Apps with Ember 2

Matthew White

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina

The team that produced this book includes:

Katharine Dvorak (editor)
Potomac Indexing, LLC (index)
Liz Welch (copyedit)
Dave Thomas (layout)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-078-3
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—October 2015

CHAPTER 5

Modeling Your Data

Ember has a lot to offer as a front-end web framework. So far, we've focused almost entirely on your app's rendering and navigation. But as you build apps, you'll eventually want to include data. Ember has a solution for that: Ember Data.

Ember Data packs many things into one small library. It offers your web client a straightforward modeling approach to include simple records in your app. It reads from RESTful services, caches the data it loads, and persists these records back to the server.

In this chapter you'll learn how to use Ember Data to access data from a RESTful service. We'll focus on an optimally designed RESTful service that requires little to no configuration from our app. By looking at all of the data-related code we've added to EmberNote so far, you'll see how Ember Data works. Let's start by looking at the model classes we've created so far.

Define Your Models

An Ember Data model class represents a record in some sort of persistent storage. Because we're working on the web, it's likely that this persistent storage is accessed via a RESTful API. An Ember Data record is an instance of a model class that actually holds data; a model is simply the definition of that record. The most important thing about a record is the data it maintains. Let's look at how to define what data is maintained by a record.

Defining a Model's Fields

As with each of the other classes we've created, we define models by first running an Ember CLI command to get a stub version of the class. In *Adding a Data Service,* on page ?, we ran the following command to generate our `User` model class.

```
$ ember generate model user
```

The stub version of this class had no fields. We'll add them so that our class looks like this:

ch5/ember-note/app/models/user.js
```
import DS from 'ember-data';

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

With Ember Data, your model definition is very simple. Your model has a set of fields that you define as key-value pairs in your class. The key is how you will reference the data from within your templates (that is, user.name).

When you define a field, you have a few options for defining the field type. If you're adding a field to hold a single value, you'll use DS.attr. Calling this function creates a field in your model. Call this function with no parameters, and the model will simply accept whatever data type is passed to it.

Alternatively, you can give it a type, known as a Transform, and Ember Data will take the raw data provided by your API and try to make it fit the type you provide. That's what our user.js example does. Ember Data offers four available default choices: string, number, Boolean, and date. If needed, you can define your own Transform implementations, but we'll get by with the basics for now.

You can also define a default value for your field. For example, if your call to DS.attr looks like this:

```
name: DS.attr("string", {defaultValue: "Matt"})
```

the value Matt will be used if a given record doesn't have a value for name.

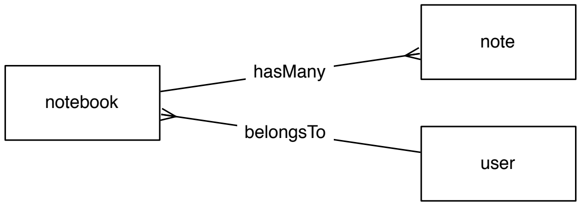
Defining Model Relationships

Your model classes might have fields that contain a record, or even a list of records, rather than a basic data type. Ember Data makes this possible through the use of two constructs: DS.hasMany and DS.belongsTo. To see an example of this, let's look at the notebook.js file:

ch5/ember-note/app/models/notebook.js
```
import DS from 'ember-data';

export default DS.Model.extend({
  title: DS.attr('string'),
  user: DS.belongsTo('user'),
  notes: DS.hasMany('note')
});
```

Both the hasMany and belongsTo functions define a relationship between the calling class and another model definition. The argument to this function is the name of the other model's class. In the notebook model example, we reference the user model from our belongsTo function and the note model from our hasMany function. Semantically, this makes sense: a notebook belongs to a user and has many notes.



It is often desirable to traverse model relationships in both directions. For example, from the perspective of the note, you could include a belongsTo relationship back to the notebook. And from the user, you could include a hasMany relationship to the notebook.

Of course, the cardinality of model relationships isn't always a one-to-many or a many-to-one. If you want a one-to-one relationship, simply create both relationships using belongsTo. For a many-to-many, both sides should be a hasMany.

Now that you have a better understanding of how to define a model class, you'll need to learn how to populate them. Let's take a look at how to load data from a server.

Load Data from RESTful Services

When using Ember to build your apps, it's very likely that your data will come from a RESTful service. A RESTful service uses HTTP as a data query and delivery mechanism. Based on HTTP requests, it can be used to return data to the calling logic, or to modify a data store in some manner.

The Ember project is very keen on adapting the best ideas from the web community into the Ember framework, and Ember Data's use of RESTful services is no exception. By simply declaring your model class, you are able to use the DS.Store object to get data from your server, assuming that the RESTful service exposes the expected API.

Although we won't dig too deeply into the implementation of a RESTful service, throughout the rest of the chapter we'll note the URL to which Ember Data makes requests. In *Use Serializers to Access Legacy APIs, on page ?*, we'll cover how to use Ember Data to access APIs that don't follow the expected conventions.

RESTful APIs

As with many notions in the programming world, there's some debate over what constitutes a RESTful API. (Often this takes the form of "Your API isn't RESTful!" "No, YOUR API isn't RESTful!") Leaving all of that aside, let's talk about how RESTful APIs are meant to be used with Ember Data.

Generally speaking, a RESTful service is a means of using a uniform resource identifier to access data and execute functions. RESTful services often, but not always, use HTTP as a transport protocol. Each unique URI and verb combination defines an action and the related record (or set of records) to use when applying that action. For example, if you want to get the note record with id 24, you can make a `GET` request of `/note/24` and expect to get the current state of the record, if it exists.

The implementation of a RESTful API is beyond the scope of this book, with the exception of the mock services we've been creating (and don't use these in a production environment, please). You might use any number of technologies to implement one, such as Ruby on Rails, Node.js, Java, or .NET. The choice is yours.

Once you've defined a model class, you'll want to be able to load data from that class's API endpoint. Before you can do that, your app will need to declare an instance of the `DS.RESTAdapter` class. Let's look at the one we declared for the EmberNote project:

ch5/ember-note/app/adapters/application.js
```
import DS from 'ember-data';

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

This adapter is as terse as it gets. The single function of this implementation of `RESTAdapter` is to identify the API's root URL. In general, an adapter is able to be responsible for much more than this, but we'll start with a simple one for now and cover adapters in greater detail in *Adapt to a Nonconventional API*, on page ?.

With this adapter in place, we're ready to begin querying the API for records.

Querying the API

When we first began working with Ember Data, we used it to display a list of notebooks that the logged-on user owned. As you can see from our definition, each notebook record belongs to one `user`:

ch5/ember-note/app/models/notebook.js

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

export default DS.Model.extend({
  title: DS.attr('string'),
  user: DS.belongsTo('user'),
  notes: DS.hasMany('note')
});
```

Given that model, we used the model hook of the notebooks route to load data from the API:

ch5/ember-note/app/notebooks/route.js

```
import Ember from 'ember';

export default Ember.Route.extend({
  model: function(params) {
    return this.store.query('notebook',{user: params.user_id});
  },
  actions: {
    addNotebook: function() {
      var notebook = this.store.createRecord('notebook', {
        title: this.controller.get('title'),
        user: this.controllerFor('application').get('user')
      });
      notebook.save().then(() => {
        console.log('save successful');
        this.controller.set('title',null);
        this.refresh();
      }, function() {
        console.log('save failed');
      });
    }
  }

});
```

As you'll recall from *Set Your Model*, on page ?, the model hook is used to obtain the data that will be displayed for a given route. We use the this.store object to find data within the route. The store object is an instance of DS.Store, which is the client-side data persistence store.

The find function typically takes two parameters. The first is the class name for the data type that the find function will return. The second is a list of parameters to query against. If this list is blank, all records of that type should be returned by the API. In our case, we provide the ID of the currently logged-in user, so all notebooks returned will belong to that user. This list of parameters can contain multiple values, and if it does, the API is expected to return all records that match each of the criteria provided.

If you don't know the ID of a record you want to load, you can write queries using the contents of other fields. That's the query form we're using in route.js, when we call this.store.query('notebook',{user: params.user_id}) to find the notebooks that belong to a certain user. The user's ID is a known piece of information about the notebook, but it doesn't uniquely identify the notebook. However, if you do know the record's ID, like in our edit-note example from the previous chapter, then there's another form of the find function you can use:

```
model: function(params) {
  return this.store.findRecord("note",params.id);
}
```

The biggest difference is that in this form, we're not defining which field to search. Ember Data assumes that we're searching the id field of the given record. If you want to load all records, you simply call findAll with no parameters, like this:

```
this.store.findAll("note");
```

The different forms of querying your API result in slightly different HTTP requests to your back end. The query form creates a GET request to your API that uses URL query parameters to communicate the query to the back end. For example, this.store.query("notebook", { user: 1}) will result in this request: GET "/api/notebooks?user=1". The id-based form also creates a GET request. For example, this.store.findRecord("note",1) results in GET "api/note/1".

These URLs are the default URLs generated by Ember Data. However, if your server API doesn't match what Ember Data generates, you don't need to change your server. In *Allow the Adapter to Query a Nonconventional API*, on page ?, we'll see how to use Ember Data to alter the URLs it queries against to match an existing API. This is one of the many ways Ember Data makes it easy to consume a RESTful API. Another is browser-based caching, which we'll look at now.

Using the Store

If all the DS.Store object did was to proxy access to a RESTful API, it would still be useful. But it also provides a browser-based cache of the data loaded since your user's session began. Once a record is loaded by a user via the find methods described in the previous section, that record resides in the cache, providing faster access to it.

We've worked with the store variable already to find records, but we haven't really discussed what it is. The store is your primary resource for interacting with RESTful APIs. In addition to providing the find methods, it caches the data returned by those methods and provides methods for working with the cached data. It also provides methods for modifying records, as we'll see later in this chapter. An instance of DS.Store, as represented by the store variable, is available in every instance of Ember.Route.

Ember Data provides a variety of methods for working with data once it has been loaded into the store. To begin with, if you want to get all records of a certain type that are already in the store, it provides the peekAll method:

```
var allNotes = this.store.peekAll("note");
```

This method returns an array of all note records that are currently in the store. Whenever you call a query method of any of the varieties described earlier, the results are loaded into the store and available throughout the app.

When working with a browser-side cache, you'll need to be aware of scenarios when other users may have modified cached data. The store isn't kept up to date with each change as it occurs, so it's a good idea to reload records from the server when you need to use them, if they are able to be modified by others. To do this, Ember Data provides the reload flag for findRecord and findAll. For example, if we want to make sure we have the most recent changes to a shared note record when we load our route, we would call the following:

```
model: function(params) {
  this.store.findRecord("note",params.id,{ reload:true })
}
```

Using reload:true will always ensure that you get the record from the server and include the most recent changes.

There's a second way to accomplish this, assuming you have a handle on the record. If a record has already been loaded in the cache, findRecord calls the reload method on the record instance. If you already have the record, you can do the following to refresh this object from the server:

```
record.reload();
```

DS.Store also contains a method to pull all records of a given type from the server, regardless of cache state. To do this, call the findAll method with reload:true:

```
model: function(params) {
  this.store.findAll("note",{ reload:true });
}
```

You may be thinking that the only way to guarantee you have the most recent data is to use the reload parameter. This is partially true, because the findRecord and findAll methods will return records already cached in the store, if any exist. However, DS.Store provides a way to evict all records of a given type from the store:

```
model: function(param) {
  // Before looking for my data, flush the cache
  this.store.unloadAll("note");
  // Now get my records
  return this.store.query("note",{notebook: param.notebook_id});
}
```

Calling unloadAll will result in a clean slate for the following query method to work with. This is useful when you want to be able to execute named-parameter queries against the store, while making sure that the results it returns are the most recent data from the server.

DS.Store has a number of methods for working with server APIs and for storing results of API calls in a browser-resident repository. As you'd expect, it also provides a number of ways of dealing with those records once they've been loaded. In the next section we'll look at how to use Ember Data to use the records we get from the API.