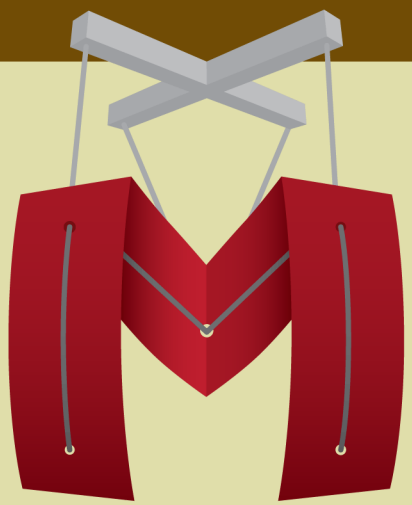
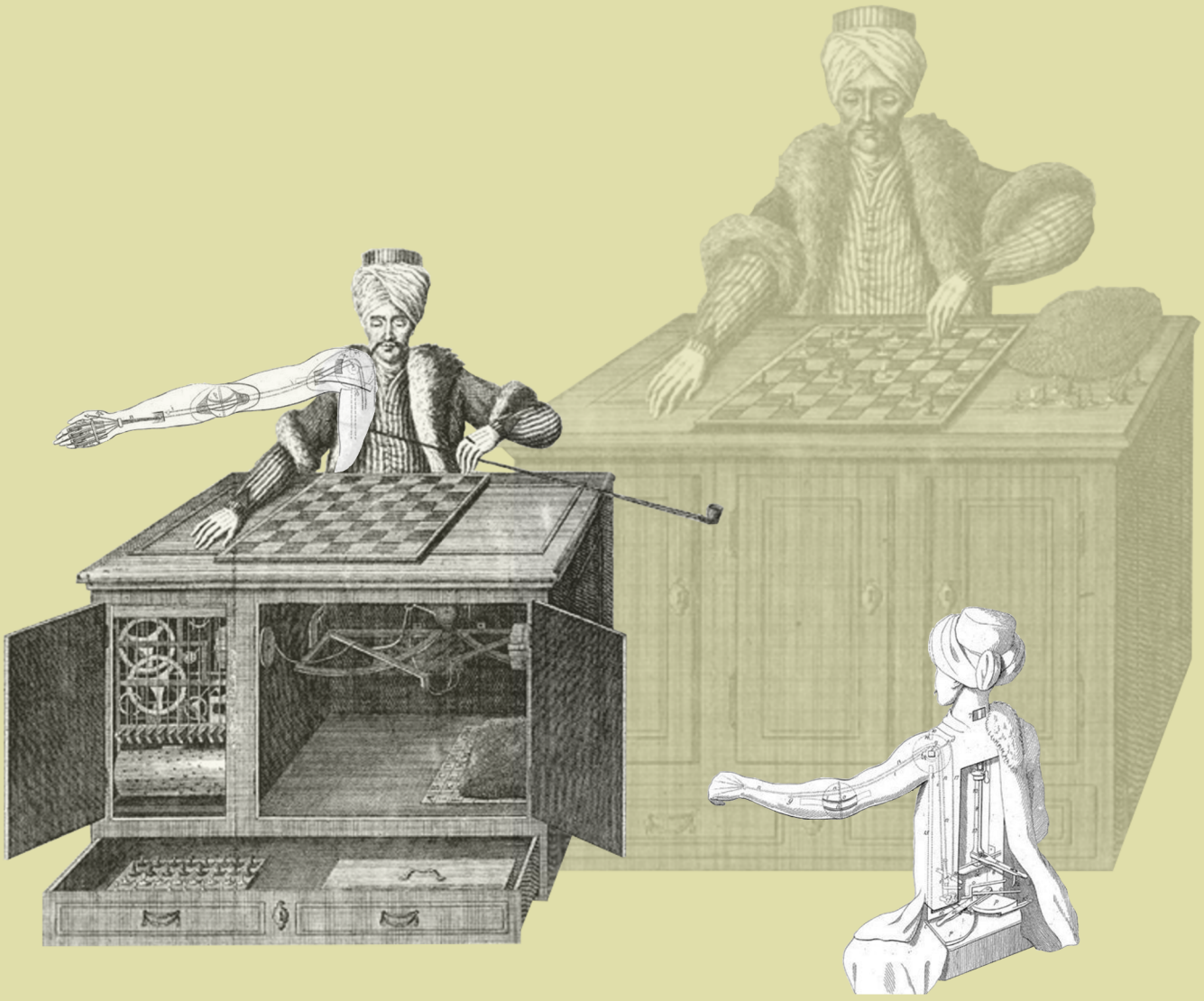
Solutions to App Development Challenges



Backbone Marionette.js



A Serious Progression

by David Sulc

Backbone.Marionette.js: A Serious Progression

David Sulc

This book is for sale at http://leanpub.com/marionette-serious-progression

This version was published on 2014-10-29

Tweet This Book!

Please help David Sulc by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Reading ”Marionette: A Serious Progression”. Check it out at
https://leanpub.com/marionette-serious-progression

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=#

Also By David Sulc

Backbone.Marionette.js: A Gentle Introduction

Structuring Backbone Code with RequireJS and Marionette Modules

Contents

Cover Credits

The cover is composed of various engravings depicting the “Mechanical Turk”, a fake chess-playing machine constructed in the late 18th century. All images are in the public domain, and were taken from the dedicated wikipedia entry[1].

[1] http://en.wikipedia.org/wiki/The_Turk

Work in Progress

This book is currently being written. Although I have a good idea of what readers want to learn thanks to the feedback from my first book[2], I'd love to hear from you! The ultimate goal, of course, is to cover the main sticking points readers run into when using Marionette in more advanced projects.

[2] https://leanpub.com/marionette-gentle-introduction

Who This Book is For

This book is for web developers who have a basic, reasonably thorough understanding of the Marionette framework. Ideally, you will have already built one or two web apps with Marionette. If you aren't yet comfortable with Marionette, you might want to check out my introductory book[3] or at least study the source code[4] of the ContactManager application (developed throughout the introductory book), as we'll be building on that web app.

This book will cover bending Backbone.Marionette.js to your will, so that your web apps remain maintainable even as you introduce advanced interaction capabilities, and must deal with sub-optimal situations (such as legacy APIs).

[3] https://leanpub.com/marionette-gentle-introduction

[4] https://github.com/davidsulc/marionette-gentle-introduction

Following Along with Git

This book is a step by step guide to building a complete Marionette.js application. As such, it's accompanied by source code in a Git repository hosted at https://github.com/davidsulc/marionette-gentle-introduction[5].

Throughout the book, as we code our app, we'll refer to commit references within the git repository like this:

Git commit with the original application:

c2f1e31eecf5cf947365e5fce32473b955f0c32f[6]

This will allow you to follow along and see exactly how the code base has changed: you can either look at that particular commit in your local copy of the git repository, or click on the link to see an online display of the code differences.

Any change in the code will affect all the following commit references, so the links in your version of the book might become desynchronized. If that's the case, make sure you update your copy of the book to get the new links. At any time, you can also see the full list of commits here[7], which should enable you to locate the commit you're looking for (the commit names match their descriptions in the book).

[5] https://github.com/davidsulc/marionette-gentle-introduction

[6] https://github.com/davidsulc/marionette-serious-progression-app/commit/c2f1e31eecf5cf947365e5fce32473b955f0c32f

[7] https://github.com/davidsulc/marionette-gentle-introduction/commits/master

Setting up

This book uses Marionette >= 2.0. If you wish to learn an earlier version of Marionette (e.g. you've inherited a project with an older version), refer to the older book version included as a zip. The code using Marionette 1.7.4 is available on Github in the `marionnette-pre-v2` branch[8].

We'll be using a remote API, implemented in Ruby on Rails[9]. Don't worry, you won't need any knowledge of Rails to use it, and will be able to focus entirely on the Marionette portion.

Get the source code for the application, by either:

- downloading the source from here[10]
- using Git to clone the repository:

```
git clone git://github.com/davidsulc/marionette-serious-progression-server.git
```

Deploying

The provided Rails application is *not* recommended for use in production, as several sub-optimal modifications had to be implemented in order to provide a better learning experience for this book. Should you wish to use Rails as your framework of choice in a production application, take a look at Agile Web Development with Rails 4[11], Rails 4 in Action[12], or ruby.railstutorial.org/[13].

Don't forget that this project will start with an empty database, so you won't see any contacts initially! You'll have to create a few of your own to start with.

[8] https://github.com/davidsulc/marionette-serious-progression-app/tree/marionette-pre-v2
[9] http://rubyonrails.org/
[10] https://github.com/davidsulc/marionette-serious-progression-server/archive/master.zip
[11] http://pragprog.com/book/rails4/agile-web-development-with-rails-4
[12] http://www.manning.com/bigg2/
[13] http://ruby.railstutorial.org/

Locally

If you want a local development environment, install Rails by following these instructions[14]. Of course, you won't need to create new project, since you'll be using the one provided above. You will, however, need to install the requisite packages by executing `bundle install` in a console, from your project's root folder.

The package list includes the "pg" gem, used for interacting with a PostgreSQL database (as used by Heroku). If you only want to deploy locally, you can either:

- make sure you have PostgreSQL installed on your machine;
- comment the line (adding a "#" at the start) starting with "gem 'pg'" in the *Gemfile* file located at the project root (or remove it).

If you're on OS X and Xcode is giving you issues installing the JSON gem, try executing this command first

```
ARCHFLAGS=-Wno-error=unused-command-line-argument-hard-error-in-future \
                                                        gem install js\
on
```

and then trying to rerun `bundle install`

You can find more on the issue here[15].

You'll also need to configure your database schema by running the following command at the prompt (again from the application's root directory):

```
rake db:migrate
```

You'll see some text scroll in your console, indicating that the various schema modifications were carried out and you'll be ready to start with the Marionette development.

Last step: start the Rails server by navigating to the project folder's root in a console, and typing in

```
rails server
```

This will start a development server running locally, and will indicate the URL to use (usually `http://localhost:3000`). If you head there, you should see a message indicating that the server is ready for you.

[14] http://guides.rubyonrails.org/getting_started.html
[15] http://stackoverflow.com/questions/22352838

Remotely

If you want a (free) remote production environment, take a look at Heroku (quick start[16], deploying an application[17]). Note: I don't get anything from Heroku for mentioning their solution. I've used them in the past and the single-step deployment is simply well-suited to our objectives (i.e. focusing on Marionette, not deployment and systems administration).

Once you've deployed the application to Heroku with `git push heroku master` (the console output will indicate the URL at which your application has been deployed), you'll also need to migrate the database[18] with `heroku run rake db:migrate`. You're now ready to start with the Marionette development.

Note that you can only deploy the `master` branch to Heroku.

Building your Own

Of course, you can also develop your own API in your favorite framework. Any behavior specifics (e.g. validation logic, return status codes) will be explained at the beginning of the chapter, and as long as you have a comparable implementation you should be able to follow along.

Using the Contact Manager Application

We'll need to copy the Contact Manager application (developed in the previous book[19]): get it here[20] and copy it into your server application's *public* folder.

Git commit with the original application:

c2f1e31eecf5cf947365e5fce32473b955f0c32f[21]

[16] https://devcenter.heroku.com/articles/quickstart
[17] https://devcenter.heroku.com/articles/getting-started-with-rails4
[18] https://devcenter.heroku.com/articles/getting-started-with-rails4#migrate-your-database
[19] https://leanpub.com/marionette-gentle-introduction
[20] https://github.com/davidsulc/marionette-serious-progression-app/archive/6d4a2fdb298bab25f9310ee6ad389b5e9b39d275.zip
[21] https://github.com/davidsulc/marionette-serious-progression-app/commit/c2f1e31eecf5cf947365e5fce32473b955f0c32f

Adapting the Application

Please make sure you're using Marionette >= 2.0, or you won't be able to follow along (version 2 introduced breaking changes). Get the file from here[22] and copy it into *assets/js/vendor/backbone.marionette.js*. If you want to use an older Marionette version, refer to the book version included in the accompanying zip file. In that case, make sure you're using Marionette >= 1.7.4, or the Behaviors chapter won't work..

Changing Underscore Template Delimiters

As it happens, Underscore templates use the same delimiters as Rails' internal templating language. This will cause issues when Rails tries to process templates intended for our Marionette application. To address this, we'll change the Underscore template delimiters (see documentation[23]):

Changing Underscore's template delimiters (assets/js/app.js)

```
ContactManager.on("before:start", function(){
  _.templateSettings = {
    interpolate: /\{\{=(.+?)\}\}/g,
    escape: /\{\{-(.+?)\}\}/g,
    evaluate: /\{\{(.+?)\}\}/g
  };
});

// ContactManager.on("start", function(){
// ...
// });
```

To achieve this, we need to specify a regular expression for each original Underscore delimiter. Here's how our new delimiters compare to the previous ones:

- `{{=...}}` replaces `<%=...%>`
- `{{-...}}` replaces `<%-...%>`
- `{{...}}` replaces `<%...%>`

[22] https://raw.githubusercontent.com/davidsulc/marionette-serious-progression-app/master/assets/js/vendor/backbone.marionette.js
[23] http://underscorejs.org/#template

As you can tell, we've put this code in a "before" initializer in our application (line 1). This means that the above code will be run right before our app starts up, which is a good time to configure Underscore just how we want it.

In the code above, we've specified all 3 possible delimiters, even though our application currently only uses one. What their uses? From Underscore's documentation[24]:

- **interpolate**: expressions that should be interpolated verbatim (i.e. their value is simply placed in the template);
- **escape**: expressions that should be inserted after being HTML escaped (to prevent XSS attacks[25]);
- **evaluate**: expressions that should be evaluated without insertion into the resulting string (e.g. an `if` condition).

Due to a brain fart (and the fact that the app was developed locally), our app was initially developped using Underscore's interpolated syntax (i.e. `{{=...}}`) when displaying data. However, note that anytime you're displaying user-manipulatable data, you should escape it to prevent XSS issues. In fact, when in doubt, **use the escape syntax (i.e. `{{-...}}`) to display data**.

With the modified delimiters in place, we still need to adapt our templates to use them:

Updating the Underscore delimiters in index.html

```
1  <!-- <script type="text/template" id="header-template"> -->
2  <!--   ... -->
3  <!-- </script> -->
4
5  <script type="text/template" id="header-link">
6    <a href="#{{- url }}">{{- name }}</a>
7  </script>
8
9  <!-- <script type="text/template" id="contact-list"> -->
10 <!--   ... -->
11 <!-- </script> -->
12
13 <script type="text/template" id="contact-list-none">
14   <td colspan="3">No contacts to display.</td>
```

[24] http://underscorejs.org/#template

[25] http://en.wikipedia.org/wiki/Cross-site_scripting

```
15  </script>
16
17  <script type="text/template" id="contact-list-item">
18    <td>{{- firstName }}</td>
19    <td>{{- lastName }}</td>
20    <td>
21      <a href="#contacts/{{- id }}" class="btn btn-small js-show">
22        <i class="icon-eye-open"></i>
23        Show
24      </a>
25      <a href="#contacts/{{- id }}/edit" class="btn btn-small js-edit">
26        <i class="icon-pencil"></i>
27        Edit
28      </a>
29      <button class="btn btn-small js-delete">
30        <i class="icon-remove"></i>
31        Delete
32      </button>
33    </td>
34  </script>
35
36  <script type="text/template" id="missing-contact-view">
37    <div class="alert alert-error">This contact doesn't exist !</div>
38  </script>
39
40  <script type="text/template" id="contact-view">
41    <h1>{{- firstName }} {{- lastName }}</h1>
42    <a href="#contacts/{{- id }}/edit" class="btn btn-small js-edit">
43      <i class="icon-pencil"></i>
44      Edit this contact
45    </a>
46    <p><strong>Phone number:</strong> {{- phoneNumber }}</p>
47  </script>
48
49  <script type="text/template" id="loading-view">
50    <h1>{{- title }}</h1>
51    <p>{{- message }}</p>
52    <div id="spinner"></div>
53  </script>
54
55  <script type="text/template" id="contact-form">
56    <form>
```

```
57      <div class="control-group">
58        <label for="contact-firstName" class="control-label">First name:</label>
59        <input id="contact-firstName" name="firstName"
60                                          type="text" value="{{- firstName }}"/>
61      </div>
62      <div class="control-group">
63        <label for="contact-lastName" class="control-label">Last name:</label>
64        <input id="contact-lastName" name="lastName"
65                                          type="text" value="{{- lastName }}"/>
66      </div>
67      <div class="control-group">
68        <label for="contact-phoneNumber" class="control-label">Phone number:</labe\
69  l>
70        <input id="contact-phoneNumber" name="phoneNumber"
71                                          type="text" value="{{- phoneNumber }}"/>
72      </div>
73      <button class="btn js-submit">Save</button>
74    </form>
75  </script>
```

Using a Remote API

Now that our server app won't get confused with the templates used by our Marionette app, let's start using the API it provides. To do so, remove the local storage configuration lines from the Contact entities:

Removing the local storage configuration from contact entities (assets/js/entities/contact.js)

```
1  // Remove this line
2  // Entities.configureStorage(Entities.Contact);
3
4  Entities.ContactCollection = Backbone.Collection.extend({
5    url: "contacts",
6    model: Entities.Contact,
7    comparator: "firstName"
8  });
9
10 // Remove this line
11 // Entities.configureStorage(Entities.ContactCollection);
```

As you can see above (lines 2 and 11), we've removed[26] the lines configuring our contact entities to

[26] Technically they're only commented in the code extract displayed, but you can go ahead and remove them completely.

use local storage. This means that going forward, they will be accessing the provided `url` (see line 5, e.g.) for persistence, and therefore all information will be fetched from and saved on the server.

Since we'll no longer be using web storage, we can also go ahead and remove the associated javascript files from *index.html*:

index.html

```
<script src="./assets/js/vendor/backbone.js"></script>
<script src="./assets/js/vendor/backbone.picky.js"></script>
<script src="./assets/js/vendor/backbone.syphon.js"></script>
<!-- Remove this line -->
<!-- <script src="./assets/js/vendor/backbone.localstorage.js"></script> -->
<script src="./assets/js/vendor/backbone.marionette.js"></script>
<script src="./assets/js/vendor/spin.js"></script>
<script src="./assets/js/vendor/spin.jquery.js"></script>

<script src="./assets/js/apps/config/marionette/regions/dialog.js"></script>
<script src="./assets/js/app.js"></script>
<!-- Remove this line -->
<!-- <script src="./assets/js/apps/config/storage/localstorage.js"></script> -->
<script src="./assets/js/entities/common.js"></script>
<script src="./assets/js/entities/header.js"></script>
<script src="./assets/js/entities/contact.js"></script>
```

Let's go to URL "#contacts" and see what hapens. Within the web console (e.g. Firebug), you'll see that there's an API error indicating that each contact is unknown: `404 Not Found`. Why is this?

Let's consider how Backbone works with remote APIs: each time we execute a model's `save` method, Backbone fires off a call to the RESTful API located at the endpoint we indicate with the model's `url` attribute. Here's the code we currently have:

assets/js/entities/contact.js

```
Entities.ContactCollection = Backbone.Collection.extend({
  url: "contacts",
  model: Entities.Contact,
  comparator: "firstName"
});

var initializeContacts = function(){
  contacts = new Entities.ContactCollection([
    { id:1, firstName: "Alice", lastName: "Arten", phoneNumber: "555-0184" },
    { id:2, firstName: "Bob", lastName: "Brigham", phoneNumber: "555-0163" },
```

```
11      { id:3, firstName: "Charlie", lastName: "Campbell", phoneNumber: "555-0129" }
12    ]);
13    contacts.forEach(function(contact){
14      contact.save();
15    });
16    return contacts.models;
17  };
```

On line 14, we call the `save` method on each model instance, which prompts Backbone to fire off a call to the remote API. As you may know, RESTful APIs typically map HTTP verbs as follows:

- GET: fetch an existing model instance
- POST: create a new model instance
- PUT: update an existing model instance
- DELETE: delete an existing model

But we never call these directly, so Backbone must be doing some magic for us behind the scenes. How does it work? First, Backbone needs to send the necessary information to the API, so it can determine which model needs to be worked with. This is achieved pretty easily: the `id` attribute is provided, which the remote endpoint then uses to manipulate the correct model instance. This technique covers fetching and deleting existing model instances, but what about saving? How does Backbone determine if it should send a POST request (to create a new model) or PUT request (to update an existing model)?

Once again, it has to do with ids, and is relatively straightforward: if the model doesn't have an id, Backbone supposes it doesn't have a server-side representation, which means it is a new model. If the model *does* have an id, it is assumed that the model exists on the server, therefore saving means updating. To sum things up, if the model instance has an `id` attribute, a PUT request is sent, otherwise a POST request is used.

The `id` attribute is essential to Backbone's syncing mechanism, so it is vital the identifying attribute can be properly determined. Therefore, if the "id" attribute isn't called `id`, you need to set it on your model by specifying an `idAttribute` value (documentation[27]).

So now we know why we've had these `404 Not Found` errors: we're creating contacts with ids on the clients side, then calling `save`. This sends a PUT request (because the model instance has an `id` attribute), but the server can't find a model with that id. Now that we're using a remote API, let's remove the initialization code:

[27] http://backbonejs.org/#Model-idAttribute

assets/js/entities/contact.js

```
// delete this function
var initializeContacts = function(){
  contacts = new Entities.ContactCollection([
    { firstName: "Alice", lastName: "Arten", phoneNumber: "555-0184" },
    { firstName: "Bob", lastName: "Brigham", phoneNumber: "555-0163" },
    { firstName: "Charlie", lastName: "Campbell", phoneNumber: "555-0129" }
  ]);
  contacts.forEach(function(contact){
    contact.save();
  });
  return contacts.models;
};
```

Since we no longer have an `initializeContacts` function, we need to adapt the rest of our code:

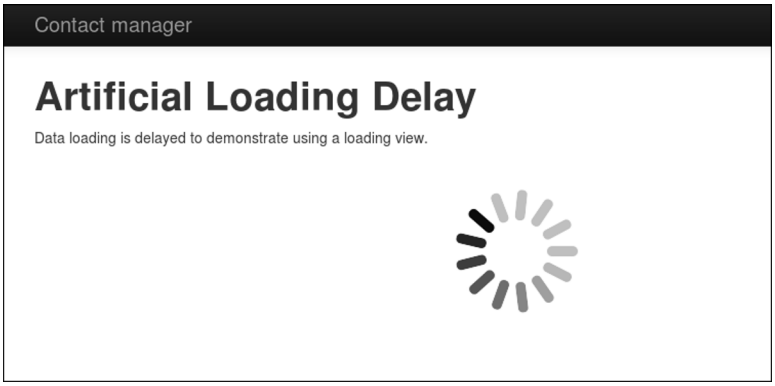
assets/js/entities/contact.js

```
var API = {
  getContactEntities: function(){
    var contacts = new Entities.ContactCollection();
    var defer = $.Deferred();
    contacts.fetch({
      success: function(data){
        defer.resolve(data);
      }
    });
    // delete these lines
    var promise = defer.promise();
    $.when(promise).done(function(contacts){
      if(contacts.length === 0){
        // if we don't have any contacts yet, create some for convenience
        var models = initializeContacts();
        contacts.reset(models);
      }
    });
    return promise;

    // return the promise
    return defer.promise();
  },
```

```
24
25    // edited for brevity
```

Don't forget to add line 22!

With our "list" action now working, let's try displaying a contact. We can see our "loading" view, due to the artificial delay still present in our application.



Our loading view

Let's remove that artificial delay (lines 4 and 13 removed):

Fetching a contact with an artificial delay (assets/js/entities/contact.js)

```
1  getContactEntity: function(contactId){
2    var contact = new Entities.Contact({id: contactId});
3    var defer = $.Deferred();
4    setTimeout(function(){
5      contact.fetch({
6        success: function(data){
7          defer.resolve(data);
8        },
9        error: function(data){
10         defer.resolve(undefined);
11       }
12     });
13   }, 2000);
14   return defer.promise();
15 }
```

And here's the same code without an artificial delay:

Fetching a contact without artificial delay (assets/js/entities/contact.js)

```
getContactEntity: function(contactId){
  var contact = new Entities.Contact({id: contactId});
  var defer = $.Deferred();
  contact.fetch({
    success: function(data){
      defer.resolve(data);
    },
    error: function(data){
      defer.resolve(undefined);
    }
  });
  return defer.promise();
}
```

Since we no longer have an artificial loading delay, let's adapt our loading view to no longer display a message mentioning an artificial loading delay:

assets/js/apps/contacts/show/show_controller.js

```
ContactManager.module("ContactsApp.Show", function(Show, ContactManager,
                                                   Backbone, Marionette, $, _){
  Show.Controller = {
    showContact: function(id){
      // add this line to use the default loading message
      var loadingView = new ContactManager.Common.Views.Loading();
      // remove these lines as they're no longer needed
      //var loadingView = new ContactManager.Common.Views.Loading({
      //  title: "Artificial Loading Delay",
      //  message: "Data loading is delayed to demonstrate using a loading view."
      //});
      ContactManager.mainRegion.show(loadingView);

  // edited for brevity
```

assets/js/apps/contacts/edit/edit_controller.js

```
ContactManager.module("ContactsApp.Edit", function(Edit, ContactManager,
                                                   Backbone, Marionette, $, _){
  Edit.Controller = {
    editContact: function(id){
      // add this line to use the default loading message
      var loadingView = new ContactManager.Common.Views.Loading();
      // remove these lines as they're no longer needed
      //var loadingView = new ContactManager.Common.Views.Loading({
      //  title: "Artificial Loading Delay",
      //  message: "Data loading is delayed to demonstrate using a loading view."
      //});
      ContactManager.mainRegion.show(loadingView);

  // edited for brevity
```

So far, so good! Let's now create a new contact: again, we get the `404 Not Found` error returning from a PUT call. Let's take a look at our code to determine why that's happening. Here's the code getting executed when the form is submitted to create a new contact:

assets/js/apps/contacts/list_controller.js

```
view.on("form:submit", function(data){
  if(contacts.length > 0){
    var highestId = contacts.max(function(c){ return c.id; }).get("id");
    data.id = highestId + 1;
  }
  else{
    data.id = 1;
  }
  if(newContact.save(data)){
    contacts.add(newContact);
    // code truncated for brevity
```

You may have noted that the contact gets added to the list view anyway, but disappears on page refresh. This will be fixed and explained below.

As you can plainly see on lines 2-8, we're manually adding a value to the `id` property. This should no longer be the case when working with a remote API, since the server should be the one assigning ids as model instances get persisted. Let's change the code to no longer specify an `id` value:

assets/js/apps/contacts/list_controller.js

```
view.on("form:submit", function(data){
  if(newContact.save(data)){
    contacts.add(newContact);
    // code truncated for brevity
```

When we try to create a new contact this time, we can see a POST request is being fired off correctly. But then we get a javascript error:

```
ReferenceError: id is not defined
```

But interestingly, if we refresh the "#contacts" page, the new contact appears... So where is this error coming from? Let's consider what happens when a new contact gets added on the list page:

1. A POST request is sent to the API;
2. The new model instance is added to the collection;
3. The collection/composite view rerenders the collection (because the contents changed)

Somewhere around the first and second steps, the API returns with the response data. In the last step, each model is rendered with the defined item view. Let's take a look at its associated template:

index.html

```
<script type="text/template" id="contact-list-item">
  <td>{{- firstName }}</td>
  <td>{{- lastName }}</td>
  <td>
    <a href="#contacts/{{- id }}" class="btn btn-small js-show">
      <i class="icon-eye-open"></i>
      Show
    </a>
    <a href="#contacts/{{- id }}/edit" class="btn btn-small js-edit">
      <i class="icon-pencil"></i>
      Edit
    </a>
    <button class="btn btn-small js-delete">
      <i class="icon-remove"></i>
      Delete
    </button>
  </td>
</script>
```

As you can see on lines 5 and 9, we refer to the `id` attribute to create the appropriate links. But at this stage, we don't have an `id` value: we've sent the creation data to the API, but we haven't got an answer back yet, and therefore don't have an id to use. How can we fix this? By using a `success` callback to wait for the server response before proceeding:

assets/js/apps/contacts/list_controller.js

```
1  view.on("form:submit", function(data){
2    var contactSaved = newContact.save(data, {
3      success: function(){
4        contacts.add(newContact);
5        view.trigger("dialog:close");
6        var newContactView = contactsListView.children.findByModel(newContact);
7        // check whether the new contact view is displayed (it could be
8        // invisible due to the current filter criterion)
9        if(newContactView){
10         newContactView.flash("success");
11       }
12     }
13   });
14   if( ! contactSaved){
15     view.triggerMethod("form:data:invalid", newContact.validationError);
16   }
17 });
```

On line 2, we save the return value from the `save` call. If Backbone is unable to save the model due to validation errors, this value will be `false` and will trigger the validation errors getting displayed (lines 14-16). If the `save` call is successful, the callback on lines 3-13 waits for the API to respond and handles the contact display.

We can't use the `error` callback to display errors in this case, because it is only triggered by API errors (not client-side validation errors), and at this time we're not using them yet. Responding to API errors will be covered later.

This also addresses the case above where the contact would be created and added to the list view even though the server threw an error (and would then disappear on page refresh). Before, the code didn't wait for the API response and therefore hadn't yet received an error before deciding to proceed (including rendering a new item view for the model, even though the model hadn't been successfully persisted on the server). With the new version forcing it to wait, this is no longer an issue.

Editing and deleting contacts already work properly, so our app is now functional just as we had it when it was configured to use local storage.

Git commit adapting the app to work with a remote API:

af922a422d1f0a6a678d0d7e309556681650acd8[28]

[28] https://github.com/davidsulc/marionette-serious-progression-app/commit/af922a422d1f0a6a678d0d7e309556681650acd8

Dealing with Legacy APIs

In some projects, you'll probably be dealing with APIs that you can't modify, and that don't conform to Backbone's expectations. In the following pages, we'll see how we can make this "difference of opinion" as invisible as possible both to the javascript front-end.

API Properties

We'll use a "contacts_legacy" endpoint that will return contacts as a JSON object associated to the `contact` key:

```
{
    "contact": {
        "id": 5,
        "firstName": "Alice",
        "lastName": "Arten",
        "phoneNumber": "555-0184",
        "createdAt": "2013-11-12T06:04:30.415Z",
        "updatedAt": "2013-11-12T06:04:30.415Z"
    }
}
```

The `createdAt` and `updatedAt` attributes aren't necessary: we won't be using them.

This means that the data regarding our contact is no longer found in the top-level JSON object, but must be parsed from within.

In addition, the API expects provided contact data to be located within a JSON object associated to a `data` key:

```
{
    "data": {
        "firstName": "John",
        "lastName": "Doe",
        "phoneNumber": "555-8784"
    }
}
```

Rewriting a Model's `parse` Function

Let's have our `contact` entities use a legacy API by changing the appropriate attributes:

assets/js/entities/contact.js

```
Entities.Contact = Backbone.Model.extend({
  urlRoot: "contacts_legacy",

  // edited for brevity
});

Entities.ContactCollection = Backbone.Collection.extend({
  url: "contacts_legacy",
  model: Entities.Contact,
  comparator: "firstName"
});
```

Happily, Backbone lets us define a `parse` method on our model to do just what we want: specify how the data received from the API should be parsed and transformed into a JSON object that is "castable" into a model instance. Let's write it:

assets/js/entities/contact.js

```
Entities.Contact = Backbone.Model.extend({
  // edited for brevity

  parse: function(data){
    if(data.contact){
      return data.contact;
    }
    else{
      return data;
    }
```

```
11    },
12
13    // edited for brevity
```

The `parse` function's `data` argument is the data received from the API.

Since the `parse` function's role is to provide a usable JSON object that can then be turned into a model, we can also use it to enrich the data. Let's add a `fullName` property to our model:

assets/js/entities/contact.js

```
1  Entities.Contact = Backbone.Model.extend({
2    // edited for brevity
3
4    parse: function(response){
5      var data = response;
6      if(response && response.contact){
7        data = response.contact;
8      }
9      data.fullName = data.firstName + " ";
10     data.fullName += data.lastName;
11     return data;
12   },
```

We can now change the existing template displaying a given contact (e.g. at URL "#contacts/5"), in order to use this new model attribute:

assets/js/apps/contacts/show/show_view.js

```
1  <script type="text/template" id="contact-view">
2    <h1>{{- data.fullName }}</h1>
3
4    <!-- edited for brevity -->
5  </script>
```

It's important to understand that a model's `parse` function can be used for both

- cleaning up and formating data coming from the API
- enriching and preparing data for display

In other words, a model's `parse` method is a great place to "massage" the data into a format we'll be comfortable working with, e.g. renaming attributes, converting snake_case to camelCase, etc.

Great! We've got the "reading" contacts from the API working correctly. Deleting contacts also works "for free" with the legacy API, because all that is involved with model deletion is sending an HTTP DELETE request to the proper endpoint. Since we've specified the "contacts_legacy" endpoint, we're good to go.

Rewriting a Model's `toJSON` Function

If we try to edit or create a new model, we get a "505 Internal Server Error" from the server. This is because the API expects the model data to be within a `data` object, as indicated earlier.

So let's make sure we have the model represented within a "data" attribute, by adding a `toJSON` method:

assets/js/entities/contact.js

```
1  Entities.Contact = Backbone.Model.extend({
2    // edited for brevity
3
4    toJSON: function(){
5      return {
6        data: _.clone(this.attributes)
7      };
8    },
9
10   // edited for brevity
11 });
```

All model attributes will be sent to the API, whether they are persisted server-side or not. Your API therefore needs to deal with any extra attributes (such as `fullName` in our case) by either ignoring them, or interpreting them correctly. If the API raises an error when encountering unknown attributes, you will need to remove them from the attributes that get sent. You can accomplish this either by removing the extra attributes in the `toJSON` method, or as we'll see in the next chapter, by removing them in the `sync` method.

Per the documentation[29], `toJSON` is used (among others) for augmentation before being sent to the server. Therefore, next time we save, our contact data should be nicely wrapped within an object linked to the `data` attribute, just as the API expects it. But if we go to the "#contacts" URL, Underscore throws an error:

[29] http://backbonejs.org/#Model-toJSON

```
ReferenceError: firstName is not defined
```

This is because it is looking for a top-level attribute named `firstName` to insert into the template. But since our contact data is wrapped within a `data` attribute (due to our `toJSON` method), there's no such top-level attribute. Let's change our template:

index.html

```
<script type="text/template" id="contact-list-item">
  <td>{{- data.firstName }}</td>
  <td>{{- data.lastName }}</td>
  <td>
    <a href="#contacts/{{- data.id }}" class="btn btn-small js-show">
      <i class="icon-eye-open"></i>
      Show
    </a>
    <a href="#contacts/{{- data.id }}/edit" class="btn btn-small js-edit">
      <i class="icon-pencil"></i>
      Edit
    </a>
    <button class="btn btn-small js-delete">
      <i class="icon-remove"></i>
      Delete
    </button>
  </td>
</script>
```

As you can see, since `toJSON` wraps everything in a `data` attribute, we need to pass through it in the template to access our model attributes and display them. Let's adapt our form template to deal with this:

index.html

```
<script type="text/template" id="contact-form">
  <form>
    <div class="control-group">
      <label for="contact-firstName" class="control-label">First name:</label>
      <input id="contact-firstName" name="firstName"
                                    type="text" value="{{- data.firstName }}"/>
    </div>
    <div class="control-group">
      <label for="contact-lastName" class="control-label">Last name:</label>
      <input id="contact-lastName" name="lastName"
```

```
                                    type="text" value="{{- data.lastName }}"/>
    </div>
    <div class="control-group">
      <label for="contact-phoneNumber" class="control-label">Phone number:</labe\
l>
      <input id="contact-phoneNumber" name="phoneNumber"
                                    type="text" value="{{- data.phoneNumber }}"/>
    </div>
    <button class="btn js-submit">Save</button>
  </form>
</script>
```

Ideally, the API will return a full representation of the object even after a PUT request. This way, you will always receive the most up-to-date version of the server-side model: another user may have modified the same model in the meantime. In this case, Backbone will update your local model with the data received from the server. If your API doesn't return the object after a PUT request, you'll need to modify the `parse` method to deal with that case:

assets/js/entities/contact.js

```
  parse: function(response){
    var data = response;
    if(response){
      if(response.contact){
        data = response.contact;
      }
      data.fullName = data.firstName + " ";
      data.fullName += data.lastName;
      return data;
    }
    else{
      this.set({fullName: this.get("firstName") + " " + this.get("lastName")});
    }
  },
```

As you can see, if there's no data in the response from the API, we simply set the `fullName` attribute directly on the model instance.

While we're at it, we can also update our view to use the `fullName` attribute computed in the `parse` method:

assets/js/apps/contacts/edit/edit_view.js

```
ContactManager.module("ContactsApp.Edit", function(Edit, ContactManager,
                                                   Backbone, Marionette, $, _){
  Edit.Contact = ContactManager.ContactsApp.Common.Views.Form.extend({
    initialize: function(){
      this.title = "Edit " + this.model.get("fullName");
    },

    // edited for brevity
```

If we edit a model from the list view and save the changes, we can see that our data is persisted correctly. That's great, but modifying all of our templates to deal with the data wrapping isn't ideal. Instead, we can intervene at Backbone's sync layer to deal with this in a way that is completely transparent from the templates, as we'll see in the next chapter.

Git commit dealing with legacy APIs:

b9eafb476832db548de5b8e4b2c5d38bbd73d818[30]

Using Non-Standard API Endpoints

Sometimes the API you need to use has endpoints that differ from the usual REST endpoints. For instance, you'd normally use URL *contacts/1* to GET a contact instance, but let's say your API makes the resource available at *contacts/1.json* instead. Here's how you could handle that:

assets/js/entities/contact.js

```
Entities.Contact = Backbone.Model.extend({
  urlRoot: "contacts",
  url: function(){
    return this.urlRoot + "/" + this.get("id") + ".json";
  },

  // edited for brevity
});

Entities.ContactCollection = Backbone.Collection.extend({
  url: "contacts.json",
  model: Entities.Contact,
```

[30] https://github.com/davidsulc/marionette-serious-progression-app/commit/b9eafb476832db548de5b8e4b2c5d38bbd73d818

```
13    comparator: "firstName"
14  });
```

We define our collection's URL to specify the "json" extension on line 11. Usually, Backbone will determine a model's URL by adding its `id` value to the collection's URL: in this case we would get *contacts.json/1*, which isn't what we want. Instead, we need to define a `urlRoot` on line 2 so that Backbone ignores the collection when building the model's URL, and we also need the function on lines 3-5 to generate a given model's URL for us.

Rewriting sync

We will be dealing with the same API described in the previous chapter.

At the Model Level

As discussed at the end of the previous chapter, we can rewrite our contact model's sync method to deal with wrapping our data before sending it to the API, making the operation transparent to the view templates. First, let's remove the toJSON implementation we've added, since the wrapping will now be done in the sync method.

assets/js/entities/contact.js

```
ContactManager.module("Entities", function(Entities, ContactManager, Backbone,
                                                          Marionette, $, _\
){
  Entities.Contact = Backbone.Model.extend({
    urlRoot: "contacts_legacy",

    // edited for brevity

    toJSON: function(){
      return {
        data: _.clone(this.attributes)
      };
    },

    // edited for brevity
```

Now, we can move on to implementing the model's sync method. But first, let's discuss what exactly Backbone's sync is: anytime a Backbone model or collection requires an API call (e.g. save, fetch), Backbone relies on the sync function. The implementation will be searched for along the prototype chain, which means we can adapt sync's behavior in several places:

- the model instance's sync method
- the model's sync method (i.e. the "class" level)
- a parent model's sync method

- Backbone's top-level `sync` function

As you may recall, we already intervened within Backbone's top-level `sync` function: in the previous book[31], we use the localStorage plugin which rewrites Backbone's `sync` layer to use web storage (instead of a remote API) for persistence.

Right now we're going to intervene at the model layer, but we'll see example of the other options later. So, how does `sync` work? In a nutshell, it translates application data needs to HTTP verbs and API calls:

- read an existing model instance: GET request
- create a new model instance: POST request
- update an existing model instance: PUT request
- delete an existing model instance: DELETE request

If you're designing your own APIs, it's important you bear in mind the following:

- GET requests shouldn't modify data, they are essentially read-only operations
- PUT requests should send absolute data (as opposed to relative data): you should (e.g.) send "`price: 5.99`" instead of "`price: +1`". Of course, the latter is possible, but it is an incredibly bad idea and will expose you to no end of misery

Matching to the GET and DELETE verbs is pretty straightforward: they correspond to `fetch` and `destroy` model instance methods, respectively. What about POST and PUT verbs? They both match the model's `save` semantic, so Backbone uses a simple trick to determine which verb is appropriate: it assumes that the server assigns model ids, so a model with an id value will trigger a PUT call (i.e. "update"), while a model without an id will initiate a POST call (i.e. "create").

It's worth noting that although REST verbs are the preferred method of API communication, you can also use the emulateHTTP[32] and emulateJSON[33] options to communicate with legacy web servers that don't understand REST verbs or JSON encoding, respectively.

Finally, the `sync` function returns a jQuery promise[34], which lets us monitor any API request and respond as needed (as the request state evolves). For example, when creating a new model instance,

[31] https://leanpub.com/marionette-gentle-introduction
[32] http://www.backbonejs.org#Sync-emulateHTTP
[33] http://www.backbonejs.org#Sync-emulateJSON
[34] http://api.jquery.com/deferred.promise/

we will be able to wait for the API response before creating the Backbone model instance locally. This means we won't needlessly create a new model instance if the API returns errors: without waiting for the response, we'd create the model instance when sending the data to the API, and when the API returns errors, we'd have to deal with an already created object (e.g. removing the model's view, displaying an error message, etc.).

Let's get started by removing the model's `toJSON` method (lines 5-10):

assets/js/entities/contact.js

```
1  parse: function(response){
2    // edited for brevity
3  },
4
5  toJSON: function(){
6    return {
7      data: _.clone(this.attributes)
8    };
9  },
10
11 validate: function(attrs, options) {
12   // edited for brevity
```

We then define our model's `sync` behavior:

assets/js/entities/contact.js

```
1  validate: function(attrs, options) {
2    // edited for brevity
3  },
4
5  sync: function(method, model, options){
6    if(method === "create" || method === "update"){
7      _.defaults(options || (options = {}), {
8        attrs: {
9          data: model.toJSON()
10       }
11     });
12   }
13   return Backbone.Model.prototype.sync.call(this, method, model, options);
14 }
```

Since we're adding the `sync` function at the end, don't forget to add the comma on line 3!

What's going on here? Let's first consider our API's behavior:

- it wraps our contact data within a `contact` object when it receives a GET request (which we unwrap with the model's `parse` method here);
- it doesn't require anything special other than a DELETE to remove a contact;
- it requires data to be wrapped in a `data` object when creating or updating a contact (i.e. POST and PUT requests).

The only reason we needed `toJSON` previously was to serialize a contact's attributes before they were sent to the API, so now we only need to interfere with `sync`'s default behavior in those same cases: creating a new model instance, and updating an existing one. Here's what's happening in the code (which is very similar to Backbone's original `sync` code, which you can consult here[35], with the portion we're interested in here[36]):

- check if we are creating a new contact or updating an existing one (i.e. we'll make a POST or PUT request) on line 6;
- assign an empty object to `options` if no parameter is provided (line 7);
- lines 7-11: if the `options` object has no `attrs` attribute, create one containing the `data` key (needed for our API), and associate it with a copy of the model's attributes;
- finally, pass these new parameters on to the original Backbone `sync` method for models (line 13).

Backbone's `sync` function expects a typical CRUD verb as the `method` argument (see documentation[37]).

In other words, if we're creating or updating a contact, we "inject" the model's parameters within an object linked to the `data` key, and pass everything on to the default `sync` function.

Don't forget to `return` the value of the prototype's `sync` call! This value will be a jQuery promise[38] that is required for various synchronization mechanisms (e.g. waiting for the API's response before proceeding).

Now that our contact's attributes get wrapped in the `sync` method instead of `toJSON`, we can restore our templates to normal by removing the reference to `data`:

[35] http://backbonejs.org/docs/backbone.html#section-130
[36] http://backbonejs.org/docs/backbone.html#section-135
[37] http://www.backbonejs.org#Sync
[38] http://api.jquery.com/deferred.promise/

index.html

```
<script type="text/template" id="contact-list-item">
  <td>{{- firstName }}</td>
  <td>{{- lastName }}</td>
  <td>
    <a href="#contacts/{{- id }}" class="btn btn-small js-show">
      <i class="icon-eye-open"></i>
      Show
    </a>
    <a href="#contacts/{{- id }}/edit" class="btn btn-small js-edit">
      <i class="icon-pencil"></i>
      Edit
    </a>
    <button class="btn btn-small js-delete">
      <i class="icon-remove"></i>
      Delete
    </button>
  </td>
</script>

<!-- edited for brevity -->

<script type="text/template" id="contact-form">
  <form>
    <div class="control-group">
      <label for="contact-firstName" class="control-label">First name:</label>
      <input id="contact-firstName" name="firstName"
                                      type="text" value="{{- firstName }}"/>
    </div>
    <div class="control-group">
      <label for="contact-lastName" class="control-label">Last name:</label>
      <input id="contact-lastName" name="lastName"
                                      type="text" value="{{- lastName }}"/>
    </div>
    <div class="control-group">
      <label for="contact-phoneNumber" class="control-label">Phone number:</labe\
l>
      <input id="contact-phoneNumber" name="phoneNumber"
                                      type="text" value="{{- phoneNumber }}"/>
    </div>
    <button class="btn js-submit">Save</button>
  </form>
```

```
42 </script>
```

In the Model's Parent Object

Obviously, if other model types also need to have their attributes wrapped in a `data` object, it would be bad practice to copy the above `sync` method in each of them. Instead, the modified `sync` function should be written once in a "base model" which gets extended by other models (e.g. "contacts"). Let's add our base model:

assets/js/entities/common.js

```
1  ContactManager.module("Entities", function(Entities, ContactManager,
2                                              Backbone, Marionette, $, _){
3    Entities.FilteredCollection = function(options){
4      // edited for brevity
5    };
6
7    Entities.BaseModel = Backbone.Model.extend({
8      sync: function(method, model, options){
9        if(method === "create" || method === "update"){
10         _.defaults(options || (options = {}), {
11           attrs: {
12             data: model.toJSON()
13           }
14         });
15       }
16       return Backbone.Model.prototype.sync.call(this, method, model, options);
17     }
18   });
19 });
```

Since our base model on line 7 is itself extending from Backbone's `Model`, any model inheriting from our "base model" will also inherit the standard model functionality (unless it has been overwritten). With our base model in place, all we have to do is extend from it and remove the `sync` function:

assets/js/entities?contact.js

```
1  ContactManager.module("Entities", function(Entities, ContactManager,
2                                              Backbone, Marionette, $, _){
3    Entities.Contact = Entities.BaseModel.extend({
4
5      // edited for brevity
6
7      validate: function(attrs, options) {
8        // edited for brevity
9      }
10
11     sync: function(method, model, options){
12       if(method === "create" || method === "update"){
13         _.defaults(options || (options = {}), {
14           attrs: {
15             data: model.toJSON()
16           }
17         });
18       }
19       return Backbone.Model.prototype.sync.call(this, method, model, options);
20     }
21   });
22
23   // edited for brevity
```

Besides removing the `sync` function, we also need to change the inheritance by extending form our base model on line 3.

Don't forget to remove the trailing comma (line 8).

Our app still works correctly by wrapping model attributes in a `data` object before sending them to the API. However, we can also add some customization at the model's `sync` level. In our case, we just want to output a simple message to the console:

assets/js/entities/contact.js

```
1  Entities.Contact = Entities.BaseModel.extend({
2    // edited for brevity
3
4    validate: function(attrs, options) {
5      // edited for brevity
6    },
7
8    sync: function(method, model, options){
9      console.log("Contact's sync function called.");
10
11     return Entities.BaseModel.prototype.sync.call(this, method, model, options);
12   }
13 });
```

Again, don't forget the comma on line 6.

Git commit rewriting sync:

dd9ad60a5655445a29017b7f214d53a86e162a35[39]

All we do here is perform our "contact model" customization of the sync function, then call the parent's implementation. This allows us to encapsulate common behavior at the "base model" layer, and continue having specific customizations at the model level. Naturally, these customizations can also be implemented at the more specific level (by defining sync on a model *instance*) or at the more generalized level (by overwriting Backbone.sync itself):

Examples of further sync customization

```
1  // customize the top-level sync function
2  Backbone.sync = function(method, model, options){
3    // your customizations here
4  };
5
6  var myContact = new ContactManager.Entities.Contact();
7  myContact.sync = function(method, model, options){
8    // your customizations here
9  };
```

[39] https://github.com/davidsulc/marionette-serious-progression-app/commit/dd9ad60a5655445a29017b7f214d53a86e162a35

Exercise: Dealing with Hyphenated API Attributes

Some APIs return attribute names that are hyphenated, for example:

```
{
    "contact": {
        "id": 5,
        "avatar-url": "http://example.com/123.png",
        "firstName": "Alice",
        "lastName": "Arten",
        "phoneNumber": "555-0184",
        "createdAt": "2013-11-12T06:04:30.415Z",
        "updatedAt": "2013-11-12T06:04:30.415Z"
    }
}
```

Think about how you could deal with the `avatar-url` attribute so you can display it in a contact's "show" template:

Displaying the `avatar-url` attribute (index.html)

```
1  <script type="text/template" id="contact-view">
2    <h1>{{- fullName }}</h1><br/>
3    <a href="#contacts/{{- id }}/edit" class="btn btn-small js-edit">
4      <i class="icon-pencil"></i>
5      Edit this contact
6    </a>
7    <p><strong>Avatar:</strong> {{- avatar-url }}</p>
8    <p><strong>Phone number:</strong> {{- phoneNumber }}</p>
9  </script>
```

The attempt on line 7 won't work: you'll get a "ReferenceError: `avatar` is not defined". This is because what you have is

```
{{- avatar-url }}
```

But what the template sees is

```
{{- avatar - url }}
```

In other words, it's trying to subtract the `url` attribute value from the `avatar` attribute, and then display the result.

Try and come up with a few options, as well as their advantages and disadvantages. If you're unable to, take a look at the next page for some hints. Note that our legacy API at "contacts_legacy" already sends an `avatar-uri` attribute for your use.

Hints

There are 3 main approaches to process the hyphenated attribute:

- the view's `serializeData` method (see documentation[40]) to make the data accessible via an attribute with a different name;
- the model's `parse` method to change the attribute names as the data comes in;
- the model's `toJSON` method to change the model's serialized representation.

Try to implement a solution with each of these, and we'll discuss the pros and cons of each.

[40] https://github.com/marionettejs/backbone.marionette/blob/master/docs/marionette.itemview.md#itemview-serializedata

Solution

Each solution is independent: when implementing one solution, we'll need to remove the code used in another solution. However, each solution uses the same template modification:

Displaying the new `avatarUrl` attribute (index.html)

```
1  <script type="text/template" id="contact-view">
2    <h1>{{- fullName }}</h1><br/>
3    <a href="#contacts/{{- id }}/edit" class="btn btn-small js-edit">
4      <i class="icon-pencil"></i>
5      Edit this contact
6    </a>
7    <p><strong>Avatar:</strong> {{- avatarUrl }}</p>
8    <p><strong>Phone number:</strong> {{- phoneNumber }}</p>
9  </script>
```

As you can see on line 7, we've changed the attribute name we want to display from `avatar-url` to `avatarUrl`. Each solution will then consist of making an `avatarUrl` value available to the template.

None of the code below will get included into our app. Once you've added the code and seen how it works, don't forget to remove it!

Use `serializeData`

We can use the view's `serializeData` to define what data will be accessible to our view template, and *how* it will be made accessible. In other words, we can define a new `avatarUrl` attribute that will be provided to the template:

assets/js/apps/contacts/show/show_view.js

```
1  Show.Contact = Marionette.ItemView.extend({
2    // edited for brevity
3
4    editClicked: function(e){
5      // edited for brevity
6    },
7
8    serializeData: function(){
9      var attr = _.clone(this.model.attributes);
10     attr.avatarUrl = attr['avatar-url'];
```

```
11      delete attr['avatar-url']; // this is optional
12      return attr;
13    }
14  });
```

As you can see on lines 8-13, we create an `attr` object to contain the data we will provide to the view, and start by filling it with the model's cloned attributes. Then, we add a new `avatarUrl` attribute (which doesn't have a hyphen), and copy the data from the hyphenated url (line 10). Once that is done, we can delete the original `avatar-url` property (line 11), although this is optional: if you don't delete it, your object will simply have both `avatarUrl` and `avatar-url` defined. Finally, we return this object so it can be accessed from the view template.

We clone the model's attributes as a best practice: this avoids any attribute modifications (e.g. from within the template) from ending up in the model.

Git commit using `serializeData`:

7c3be50e6b9a7af44c34bd0bcef1c6d0b4fcce30[41]

Discussion

Using `serializeData` is useful when all you need is to display some extra (or modified) data within a view. However, if the "extra" data gets modified, you'll need to ensure the proper attribute names and values get sent to the API. In other words, if we make the `avatar-url` attribute available as `avatarUrl` in the view and it gets updated, we'll need to send the new value within `avatar-url` to the API (since that's the name it expects), or the attribute won't get updated.

In addition, having the same data getting "enriched" via `serializeData` in multiple views tends to be a code smell indicating the data needs to be pre-processed at a higher, centralized level. Otherwise, you'll end up duplicating the `serializeData` function in multiple views. If pre-processing data at a higher level isn't an option, you should at the very least define the logic in a single place, then pass the function reference:

[41] https://github.com/davidsulc/marionette-serious-progression-app/commit/7c3be50e6b9a7af44c34bd0bcef1c6d0b4fcce30

Passing the `serializeData` function as a reference

```
1  // define the logic at a sufficiently high level (possibly at the sub-app level)
2  var mySerializeData = function(){
3    return {
4      example: "my data"
5    }
6  };
7
8  // then provide it to your view
9  MyView = Marionette.ItemView.extend({
10   serializeData: mySerializeData
11 });
```

Use `parse`

Since `parse` processes the data from the API before it reaches the model instance, we can use it to adapt our hyphenated attribute:

Replacing hyphenated attributes with `parse` (assets/js/entities/contact.js)

```
1  Entities.Contact = Entities.BaseModel.extend({
2    // edited for brevity
3
4    parse: function(response){
5      var data = response;
6      if(response && response.contact){
7        data = response.contact;
8      }
9      data.avatarUrl = data['avatar-url'];
10     delete data['avatar-url']; // this is optional
11     data.fullName = data.firstName + " ";
12     data.fullName += data.lastName;
13     return data;
14   },
15
16   // edited for brevity
```

Lines 9-10 replace the hyphenated attribute with one we can use in the templates without issues.

Git commit using `parse`:

7fc64b8fbe18c1fd8a841976afd452f78db30ab1[42]

[42] https://github.com/davidsulc/marionette-serious-progression-app/commit/7fc64b8fbe18c1fd8a841976afd452f78db30ab1

Discussion

As above, `parse` is useful to process read-only data. If a renamed attribute gets modified, it will have to be properly sent to the API when saving (i.e. send the data with the original hyphenated attribute name). The advantage this method has over `serializeData` is that it is centralized and therefore the `avatarUrl` attribute will be available from within the entire application (not just the single view where `serializeData` has been defined to manage the problem). Be aware, however, that there is a performance penalty if there are many hyphenated attributes getting replaced, especially if very few are actually needed at any given time. In those cases, you'd be better off using `serializeData`.

Use `toJSON`

We can use `toJSON` to add a new `avatarUrl` attribute mirroring `avatar-url` each time the model's attributes are serialized:

assets/js/entities/contact.js

```
1  Entities.Contact = Entities.BaseModel.extend({
2    // edited for brevity
3
4    validate: function(attrs, options) {
5      // edited for brevity
6    },
7
8    toJSON: function(){
9      var data = _.clone(this.attributes);
10     data.avatarUrl = data['avatar-url'];
11     delete data['avatar-url']; // this is optional
12     return data;
13   },
14
15   // edited for brevity
```

Don't forget we've restored the `parse` method to its original version: we no longer interact with `avatar-url` within the `parse` function.

Git commit using `toJSON`:

09301d379cba9ce93d1c2eb51999a5af5309cb08[43]

[43] https://github.com/davidsulc/marionette-serious-progression-app/commit/09301d379cba9ce93d1c2eb51999a5af5309cb08

Discussion

Whereas the `parse` function processes data as soon as it is received from the API, the `toJSON` method is executed when the model needs to be serialized. This happens when the model attributes get sent to the template, but also when they are sent to the API for persistence. In addition, `parse` gets called only *once* each time data is received, but `toJSON` gets called *each time* the model attributes need to be serialized. Just like the `parse` solution, `toJSON` will have performance issues if there are many hyphenated attributes, and using `serializeData` might be a better fit in those cases.

Another difference to be aware of between the `parse` and `toJSON` implementations: our `parse` code above removes the `avatar-url` from the attribute received by the model (from the API), and therefore the model never has access to the hyphenated attribute. The `toJSON` solution, on the other hand, removes the hyphenated attribute from the ones that get serialized. In other words, `avatar-url` is *still available* on the model and never gets removed.

It's also important to be aware that `toJSON` is used to serialize model attributes *before they are sent to the API*. Therefore, the code above will send `avatarUrl` to the API (e.g. when saving a model instance), but `avatar-url` will never be sent. This behavior can still be useful if we're retrieving data from an API using hyphens, but we're then persisting data without hyphnes (on a different API).

Note that you could combine the two solutions above so that templates have attributes without hyphens, but the original hyphenated attribute remains up-to-date (especially when sent to the API):

Keeping the hyphenated attribute up-to-date

```
1  parse: function(response){
2    // as above
3  },
4
5  toJSON: function(){
6    var data = _.clone(this.attributes);
7    data['avatar-url'] = data.avatarUrl;
8    return data;
9  },
```

On line 7, we restore the `avatar-url` attribute by copying the `avatarUrl` value into it.

Restoring the Original API

Before we proceed any further, let's go back to using our normal API:

assets/js/entities/contact.js

```
ContactManager.module("Entities", function(Entities, ContactManager,
                                           Backbone, Marionette, $, _){
  Entities.Contact = Entities.BaseModel.extend({
    urlRoot: "contacts",

    // edited for brevity

  Entities.ContactCollection = Backbone.Collection.extend({
    url: "contacts",

    // edited for brevity
```

We'll no longer be displaying the "avatar url" property:

index.html

```
<script type="text/template" id="contact-view">
  <h1>{{- fullName }}</h1>
  <a href="#contacts/{{- id }}/edit" class="btn btn-small js-edit">
    <i class="icon-pencil"></i>
    Edit this contact
  </a>
  <p><strong>Avatar:</strong> {{- avatarUrl }}</p>
  <p><strong>Phone number:</strong> {{- phoneNumber }}</p>
</script>
```

Let's also make our `BaseModel`'s `sync` function stop tampering with the data before it is sent to the API:

assets/js/entities/common.js

```
Entities.BaseModel = Backbone.Model.extend({
  sync: function(method, model, options){
    if(method === "create" || method === "update"){
      _.defaults(options || (options = {}), {
        attrs: {
          data: model.toJSON()
        }
      });
    }
```

```
10      return Backbone.Model.prototype.sync.call(this, method, model, options);
11    }
12  });
```

Git commit restoring the original API:

3bb0bcdc5934b314b93828b4484fcd7913c8bd40[44]

[44] https://github.com/davidsulc/marionette-serious-progression-app/commit/3bb0bcdc5934b314b93828b4484fcd7913c8bd40

Wrapping a Third-Party API

We've seen previously that the `sync` function is used by Backbone for all API calls, and that we can modify it to fit our needs. In this chapter, we'll see how we can wrap a third-party API with the `sync` function, so that the API calls are completely transparent to our application. In other words, we'll be dealing with native Backbone models, and the various `fetch`, `save`, and other methods will transparently fire the correct API calls.

Using GitHub's API

We're going to use GitHub[45]'s API[46] to interact with git repositories. If you don't have one already, create a GitHub account (which is free) here[47].

Then, let's test the connection by listing all of my repositories (documentation[48]). Here's the code:

Testing an unauthenticated API call

```
1 $.ajax({
2   method: "GET",
3   url: "https://api.github.com/users/davidsulc/repos"
4 });
```

Authenticating

We'll now make an authenticated API call: listing *your* repositories (documentation[49]). Since we need to be authenticated, we'll need some way to prove we're a certain user, and we'll do this by using basic authentication (which sends the username and password info in the request's headers). Happily, jQuery allows us to set request headers before firing off the request:

[45] https://github.com
[46] http://developer.github.com/v3/
[47] https://github.com/join
[48] http://developer.github.com/v3/repos/#list-user-repositories
[49] http://developer.github.com/v3/repos/#list-your-repositories

Testing an authenticated API call: listing user repositories

```
1  $.ajax({
2    beforeSend: function (xhr) {
3        xhr.setRequestHeader ("Authorization", "Basic " +
4                                   btoa("USERNAME" + ":" + "PASSWORD"));
5    },
6    method: "GET",
7    url: "https://api.github.com/user/repos"
8  });
```

Don't forget to substitute your GitHub username and password on line 4.

Working with Repositories

So far, so good. Let's now move on to manipulating GitHub repositories using native Backbone models. Here's what we'd like to be able to do:

Repository interaction example

```
1  // create a repository on the server
2  var repo = new Repository({ name: "test-api-repo" });
3  repo.save();
4  // read an existing repository
5  repo.fetch();
6  // update a repository
7  repo.set({ description: "Repository created by API" });
8  repo.save();
9  // delete an existing repository
10 repo.destroy();
```

If you look at GitHub's API documentation for the create[50], read[51], update[52], and delete[53] methods, you'll find that they have pretty different signatures... This will be interesting!

[50] http://developer.github.com/v3/repos/#create
[51] http://developer.github.com/v3/repos/#get
[52] http://developer.github.com/v3/repos/#edit
[53] http://developer.github.com/v3/repos/#delete-a-repository

Of course, if you like challenges, you can go ahead and try implementing this on your own. If you do, make sure this works in your version:

```
// get an existing repository
var repo = new Repository({ name: "test-api-repo" });
repo.fetch();
repo.set({ name: "test-api-one" });
repo.set({ name: "test-api-two" });
// rename repository on the server
repo.save();
```

sync Basics

First, let's take a look at what a typical `sync` implementation structure looks like:

A simple `sync` structure

```
sync: function(method, model, options){
  switch(method){
    // configure parent's sync or API call according to method
    case "create":
      break;

    case "read":
      break;

    case "update":
      break;

    case "delete":
      break;
  };

  // depending on inheritance, call the parent's implementation, or (if at the t\
op
  // of prototype chain) call Backbone.$ directly to execute API call
}
```

Essentially, for each CRUD action, we will "translate" the intention into the appropriate API call. To get started with the actual implementation, we'll need a new `Repository` model, so let's create it:

assets/js/entities/repository.js

```
ContactManager.module("Entities", function(Entities, ContactManager, Backbone,
                                                                  Marionette, $, _\
){
  Entities.Repository = Backbone.Model.extend({
    sync: function(method, model, options){
      switch(method){
        case "create":
          break;

        case "read":
          break;

        case "update":
          break;

        case "delete":
          break;
      };

      return Backbone.Model.prototype.sync.call(this, method, model, options);
    }
  });
});
```

An we'll need to add it to the index file (line 9):

Adding our repository model (index.html)

```
<!-- edited for brevity -->

<script src="./assets/js/apps/config/marionette/regions/dialog.js"></script>
<script src="./assets/js/app.js"></script>
<script src="./assets/js/apps/config/storage/localstorage.js"></script>
<script src="./assets/js/entities/common.js"></script>
<script src="./assets/js/entities/header.js"></script>
<script src="./assets/js/entities/contact.js"></script>
<script src="./assets/js/entities/repository.js"></script>
<script src="./assets/js/common/views.js"></script>

<!-- edited for brevity -->
```

As you can see, we've simply created a new model extending from Backbone's main `Model`, and added our `sync` skeleton in it. You'll notice that on line 19, we're proxying the arguments to the original model `sync` method. The idea is to

1. alter the received arguments so they will conform to GitHub's API;
2. proxy the altered arguments to the parent `sync` implementation (which in this case is the original `sync` implementation).

Using `call` on a Prototype's Function Definition

How does the whole `call` thing on line 19 work? It helps to think of it as calling a method from a parent "class" (i.e. calling `super` in some languages). You're already familiar with `Backbone.Model`, since that's the object we extend to create new model "classes". By accessing its `prototype` attribute, we have access to the various function definitions it provides. In other words

```
Backbone.Model.prototype.sync
```

gives us access to the normal, unaltered `sync` implementation we normally use. But we still need some way to call that implementation for the current repository model instance. Luckily, javascript provides the aptly-named `call` method for this purpose:

```
Backbone.Model.prototype.sync.call(this, ...)
```

Don't forget that the `save` method also proxies to `sync`. So in the explanation below, when we're calling `myRepo.save(...)`, you can think of it as calling `myRepo.sync(...)`.

The first argument to `call` is the context to use as the value of `this` within the called function (the ... represent the list of arguments passed to the function). If you refer back to the `Repository` model code above, you can see that the location where we have this `call` line is within `Repository`, and it will be called on a repository instance (e.g. `myRepo.save()`), so *on that line*, the value of `this` refers to the current repository model instance. Therefore, when we're doing (e.g.) `myRepo.save()`, and we get to the line with

```
Backbone.Model.prototype.sync.call(this, ...)
```

we're telling javascript to call the original `sync` implementation, and use the current `this` value (i.e. `myRepo` in the example above) as the value for `this` inside that function definition. In effect, we've got javascript executing the equivalent of `myRepo.sync(...)` which is exactly what we want.

If you'd like to learn more about these subjects, take a look at documentation for

- this[54]
- prototype[55]
- call[56]
- apply[57]

Creating a New Repository

With that explanation out of the way, let's tackle creating a new repository instance using `sync`:

assets/js/entities/repository.js

```
ContactManager.module("Entities", function(Entities, ContactManager, Backbone,
                                                          Marionette, $, _\
){
  Entities.Repository = Backbone.Model.extend({
    sync: function(method, model, options){
      var config = {};
      switch(method){
        case "create":
          config = {
            beforeSend: function (xhr) {
                xhr.setRequestHeader ("Authorization",
                                      "Basic " + btoa("USERNAME" + ":" + "PASSWORD"));
            },
            method: "POST",
            url: "https://api.github.com/user/repos",
            data: JSON.stringify({
                name: "Test"
            })
          };
          break;

        case "read":
          break;
```

[54] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this
[55] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Inheritance_and_the_prototype_chain
[56] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/call
[57] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/apply

```
24
25          case "update":
26            break;
27
28          case "delete":
29            break;
30        };
31
32        // add API call configuration to the `options` object
33        options = _.extend(options, config);
34
35        return Backbone.Model.prototype.sync.call(this, method, model, options);
36      }
37    });
38  });
```

Be aware that although these values are sent over SSL, they are visible in the source code. Therefore, unless you're developing on your local machine, take the appropriate steps to ensure that any sensitive information can't be compromised. One simple possibility is to refer to `window.username` and `window.password` in your code. Then, before working with repositories, simply set those values from the console:

```
window.username = "john";`
window.password = "open up";`
```

Let's try it out! First, take a look at the list of your repositories at

https://github.com/USERNAME?tab=repositories

Now, open a javascript console on your app's *index.html* page and execute the following:

```
// instantiate a new Repository model instance
var repo = new ContactManager.Entities.Repository();
// save it on the server
repo.save();
```

You'll see the successful call execute and if you check

https://github.com/USERNAME?tab=repositories

again you'll see a new repository named "Test" (which is the hard-coded name we have in our model on line 16). So far, so good!

If the `save` call returns a "401 Unauthorized" error, you've probably forgotten to update the model code with your own username and password.

Our first API call worked like a charm, let's now improve our code so we can actually chose a repository name when we're creating it:

assets/js/entities/repository.js

```
1  // edited for brevity
2
3  case "create":
4    config = {
5      beforeSend: function (xhr) {
6          xhr.setRequestHeader ("Authorization",
7                           "Basic " + btoa("USERNAME" + ":" + "PASSWORD"));
8      },
9      method: "POST",
10     url: "https://api.github.com/user/repos",
11     data: JSON.stringify({
12         name: this.get("name")
13     })
14   };
15   break;
16
17 // edited for brevity
```

With the change on line 12 in place, we can specify the name of the repository we want to create:

```
var repo = new ContactManager.Entities.Repository({ name: "another-test" });
repo.save();
```

Check the list of your repositories again, and you'll see the new repository we've created. Let's try creating a new repository without a name:

```
var repo = new ContactManager.Entities.Repository();
repo.save();
```

You'll get an error: 422 Unprocessable Entity. This is because all the repository CRUD calls require a repository name, but we haven't provided any. To avoid a useless API call (we know it won't work, because it's missing a required attribute), we can have Backbone check our model before making the API call by defining a `validate` method (documentation[58]). The `validate` method is called before `save` is executed, so let's use it to check our repository always has a name:

assets/js/entities/repository.js

```
Entities.Repository = Backbone.Model.extend({
  validate: function(attrs, options) {
    var errors = {}
    if (! attrs.name) {
      errors.name = "can't be blank";
    }
    if( ! _.isEmpty(errors)){
      return errors;
    }
  },

  sync: function(method, model, options){

    // edited for brevity

  }
});
```

Let's try creating a repository without a name again:

```
var repo = new ContactManager.Entities.Repository();
repo.save();
```

This time around you can see that no API call was fired, and that instead the `save` call returned `false` because there were errors. We can check what those errors were by looking at the `validationError` attribute (documentation[59]):

[58] http://backbonejs.org/#Model-validate
[59] http://backbonejs.org/#Model-validationError

```
repo.validationError;
```

And we can see that the returned object is the one we've defined within our `validation` function, which looks like this:

```
{
  name: "can't be blank"
}
```

By defining the validation method, we'd be able to do something like

```
if(repo.save()){
  // do something after successful save
  console.log("Repository saved successfully.");
}
else{
  // tell the user what went wrong, correct the issue(s), etc.
  var errors = repo.validationError;
  var messages = [];
  _.each(_.pairs(errors), function(err){
    messages.push("The " + err[0] + " attribute " + err[1]);
  });
  alert(messages.join("\n"));
}
```

In other words, we'll be able to be quicker in showing the user there is an error, as well as being more helpful in indicating what the error is (so the user can correct it).

So far, we've got our repository creation working properly, and can even specify a custom name. But the API accepts many more attributes that define a repository, and calling `get` to obtain each value will quickly become cumbersome. But at the same time, it's best not to send all model attributes undiscriminately: some APIs might fail if they receive an unrecognized attribute, and in any case you don't want to risk sending potentially confidential data. Instead, we can whitelist the attributes we want to send (lines 11-14):

assets/js/entities/repository.js

```
// edited for brevity

case "create":
  config = {
    beforeSend: function (xhr) {
        xhr.setRequestHeader ("Authorization",
                              "Basic " + btoa("USERNAME" + ":" + "PASSWORD"));
    },
    method: "POST",
    url: "https://api.github.com/user/repos",
    data: JSON.stringify(model.pick("name", "description", "homepage",
      "private", "has_issues", "has_wiki", "has_downloads", "team_id",
      "auto_init", "gitignore_template"
    ))
  };
  break;

// edited for brevity
```

Fetching an Existing Repository

Let's make it possible for our model to fetch an existing repository, given its name:

assets/js/entities/repository.js

```
// edited for brevity

case "read":
  config = {
    method: "GET",
    url: "https://api.github.com/repos/USERNAME/" + model.get("name")
  };
  break;

// edited for brevity
```

Once again, don't forget to replace the `USERNAME` placeholder on line 6 with your actual username.

Although the "read" API call can be used to retrieve information on any public repository, we're scoping it to our user's repositories.

Our `config` object doesn't have a `beforeSend` attribute in this case, because there is no need to authenticate for read access to a public repository.

Let's try it out, assuming you haven't deleted the test repositories we've created above:

```
var repo = new ContactManager.Entities.Repository({ name: "another-test" });
repo.fetch();
```

As you can see, our code is functional. But let's clean it up before proceeding:

Refactoring code (assets/js/repository.js)

```
1  sync: function(method, model, options){
2    var username = "USERNAME",
3        password = "PASSWORD",
4        baseUrl = "https://api.github.com";
5
6    switch(method){
7      case "create":
8        config = {
9          beforeSend: function (xhr) {
10           xhr.setRequestHeader ("Authorization", "Basic " +
11                                       btoa(username + ":" + password));
12         },
13         method: "POST",
14         url: baseUrl + "/user/repos",
15         data: JSON.stringify(model.pick("name", "description", "homepage",
16           "private", "has_issues", "has_wiki", "has_downloads", "team_id",
17           "auto_init", "gitignore_template"
18         ))
19       };
20       break;
21
22     case "read":
23       config = {
24         method: "GET",
25         url: baseUrl + "/repos/" + username + "/" + model.get("name")
26       };
```

```
27        break;
28
29      case "update":
30        break;
31
32      case "delete":
33        break;
34    };
35
36    // add API call configuration to the `options` object
37    options = _.extend(options, config);
38
39    return Backbone.Model.prototype.sync.call(this, method, model, options);
40  }
```

Technically, you don't need to specify "GET" as the method on line 24, as it's the default value. However, in most cases being explicit is better than being implicit...

Git commit creating and fetching GitHub repositories:

4c528877a2bc2fc613e9850e99d973dedd7966af[60]

Deleting an Existing Repository

Refer to the documentation[61] and try implementing the `delete` call on your own, as an exercise.

[60] https://github.com/davidsulc/marionette-serious-progression-app/commit/4c528877a2bc2fc613e9850e99d973dedd7966af

[61] http://developer.github.com/v3/repos/#delete-a-repository

Here's the (refactored) solution:

assets/js/entities/repository.js

```
sync: function(method, model, options){
  var username = "USERNAME",
      password = "PASSWORD",
      baseUrl = "https://api.github.com",
      config = {
        beforeSend: function (xhr) {
          xhr.setRequestHeader ("Authorization", "Basic " +
                                      btoa(username + ":" + password));
        }
      };

  switch(method){
    case "create":
      config = _.extend(config, {
        method: "POST",
        url: baseUrl + "/user/repos",
        data: JSON.stringify(model.pick("name", "description", "homepage",
          "private", "has_issues", "has_wiki", "has_downloads", "team_id",
          "auto_init", "gitignore_template"
        ))
      });
      break;

    case "read":
      config = _.extend(config, {
        method: "GET",
        url: baseUrl + "/repos/" + username + "/" + model.get("name")
      });
      break;

    case "update":
      break;

    case "delete":
      config = _.extend(config, {
        method: "DELETE",
        url:  baseUrl + "/repos/" + username + "/" + model.get("name")
      });
      break;
```

```
40    };
41
42    // add API call configuration to the `options` object
43    options = _.extend(options, config);
44
45    return Backbone.Model.prototype.sync.call(this, method, model, options);
46  }
```

We're defining a basic `config` object, containing shared attributes on lines 5-10. This object is then extended by adding the appropriate additional attributes required for each API call (e.g. lines 35-38).

The "read" call is read-only and doesn't require authentication. But since sending extra data isn't an issue, we include the authentication information anyway because it makes our code cleaner. Note that the authentication params get sent over SSL: if they weren't, we would definitely avoid sending them.

We can try out our implementation with:

```
1  var repo = new ContactManager.Entities.Repository({name: "Test"});
2  repo.fetch();
3  repo.destroy();
```

We need to call `fetch` on line 2 so that our model has an `id` value. Otherwise, Backbone will consider the model to exist only on the client side and will simply call the `success` callback and return false.

Technically, you could avoid using the `fetch` call in this case by setting the `id` attribute to some value. Since the API only requires a valid name (i.e. it doesn't use the `id` value), the deletion will take place regardless of the `id` value. Just be aware that this won't work in most cases, and you'll usually have to call `fetch` before interacting with the server data.

Git commit deleting GitHub repositories:

244fdadeed7be3dd75e7575d3e726c53d067299c[62]

Editing an Existing Repository

All that's left to do to have a functional (and complete) `sync` function is to manage the "update" case. So let's get to it:

[62] https://github.com/davidsulc/marionette-serious-progression-app/commit/244fdadeed7be3dd75e7575d3e726c53d067299c

assets/js/entities/repository.js

```
// edited for brevity

case "update":
  config = _.extend(config, {
    method: "PATCH",
    url:  baseUrl + "/repos/" + username + "/" + model.get("name"),
    data: JSON.stringify(model.pick("name", "description", "homepage", "private",
      "has_issues", "has_wiki", "has_downloads", "default_branch"
    ))
  });
  break;

// edited for brevity
```

Let's try it out:

```
var repo = new ContactManager.Entities.Repository({name: "another-test"});
repo.fetch();
repo.set("description", "Repository created with a dynamic name.");
repo.save();
```

It works!

Not so fast... Let's try updating the repository name:

```
var repo = new ContactManager.Entities.Repository({name: "another-test"});
repo.fetch();
repo.set("name", "another-repo");
repo.save();
```

You need to wait for the `fetch` call to return before executing the other lines.

This time, the update doesn't work, and we get a "404 Not Found" error. Why? Because we're asking the API to update a repository named "another-repo", which doesn't exist. To correct this, we need to use the repository's previous name when making the API call:

assets/js/entities/repository.js

```
// edited for brevity

case "update":
  config = _.extend(config, {
    method: "PATCH",
    url:  baseUrl + "/repos/" + username + "/" + model.previous("name"),
    data: JSON.stringify(model.pick("name", "description", "homepage", "private",
      "has_issues", "has_wiki", "has_downloads", "default_branch"
    ))
  });
  break;

// edited for brevity
```

And now, the following works as expected:

```
var repo = new ContactManager.Entities.Repository({name: "another-test"});
repo.fetch();
repo.set("name", "another-repo");
repo.save();
```

But we're not quite done. Let's try to rename it with multiple `set` calls:

```
var repo = new ContactManager.Entities.Repository({name: "another-repo"});
repo.fetch();
repo.set("name", "my-repo");
repo.set("name", "test-repo");
repo.save();
```

And we're back nearly where we started: the API is looking for a repository named "my-repo" which doesn't exist. This is because the value the `previous` method returns is updated on each "change" event. And since each `set` call triggers a "change" event, we can't properly keep track of the repository's name on the server using `previous`.

Instead, we need to save the repository's name each time we get data back from the server. Luckily, each time a model successfully syncs with the server, a "sync" event is triggered (documentation[63]). Let's use it:

[63] http://backbonejs.org/#Events-catalog

assets/js/entities/repository.js

```
Entities.Repository = Backbone.Model.extend({
  initialize: function(){
    var self = this;
    this.on("sync", function(){
      self.set({githubName: self.get("name")}, {silent:true});
    });
  },

  validate: function(attrs, options) {
    // edited for brevity
  },

  sync: function(method, model, options){
    // edited for brevity

    switch(method){
      // edited for brevity

      case "update":
        config = _.extend(config, {
          method: "PATCH",
          url:  baseUrl + "/repos/" + username + "/" + model.get("githubName"),
          data: JSON.stringify(model.pick("name", "description", "homepage",
            "private", "has_issues", "has_wiki", "has_downloads", "default_branc\
h"
          ))
        });
        break;

      // edited for brevity
    };

    // add API call configuration to the `options` object
    options = _.extend(options, config);

    return Backbone.Model.prototype.sync.call(this, method, model, options);
  }
});
```

Now that we rely on the `githubName` attribute which is only set when the model syncs with the server, you *must* call `fetch` before attempting to modify a model instance on the server side.

On lines 2-7, we create an event listener for each model instance: when it syncs with the server, it sets the `githubName` attribute to the repository's name on the server. Then, on line 22 we use that attribute in our API call.

We pass the `silent: true` option to the `set` call on line 5 so that the model won't trigger a "change" event: setting the `githubName` attribute should happen behind the scenes, without the model needing to know about it (or react to it).

With this change in place, we can finally do the following:

```
var repo = new ContactManager.Entities.Repository({name: "another-repo"});
repo.fetch();
repo.set("name", "my-repo");
repo.set("name", "test-repo");
repo.save();
```

Naturally, we should also use the `githubName` when deleting a repository, so it will always locate the correct repository to delete:

assets/js/entities/repository.js

```
1  // edited for brevity
2
3  case "delete":
4    config = _.extend(config, {
5      method: "DELETE",
6      url:  baseUrl + "/repos/" + username + "/" + model.get("githubName")
7    });
8    break;
9
10 // edited for brevity
```

Using the `urlRoot` and `url` Properties

Here's the current state of our `sync` function:

assets/js/entities/repository.js

```
sync: function(method, model, options){
  var username = "USERNAME",
      password = "PASSWORD",
      baseUrl = "https://api.github.com",
      config = {
        beforeSend: function (xhr) {
          xhr.setRequestHeader ("Authorization", "Basic " +
                                      btoa(username + ":" + password));
        }
      };

  switch(method){
    case "create":
      config = _.extend(config, {
        method: "POST",
        url: baseUrl + "/user/repos",
        data: JSON.stringify(model.pick("name", "description", "homepage",
          "private", "has_issues", "has_wiki", "has_downloads", "team_id",
          "auto_init", "gitignore_template"
        ))
      });
      break;

    case "read":
      config = _.extend(config, {
        method: "GET",
        url: baseUrl + "/repos/" + username + "/" + model.get("name")
      });
      break;

    case "update":
      config = _.extend(config, {
        method: "PATCH",
        url:  baseUrl + "/repos/" + username + "/" + model.get("githubName"),
        data: JSON.stringify(model.pick("name", "description", "homepage",
          "private", "has_issues", "has_wiki", "has_downloads", "default_branch"
        ))
      });
      break;

    case "delete":
```

```
42        config = _.extend(config, {
43          method: "DELETE",
44          url:  baseUrl + "/repos/" + username + "/" + model.get("githubName")
45        });
46        break;
47    };
48
49    // add API call configuration to the `options` object
50    options = _.extend(options, config);
51
52    return Backbone.Model.prototype.sync.call(this, method, model, options);
53  }
```

As you can see, we're making use of a `baseUrl` value (line 4), and using it to generate a `url` value to use as the API endpoint (e.g. line 16). But since we can define `urlRoot` and `url` properties/functions on models, we should probably use those to clean up our code.

Let's start by defining the `urlRoot` property:

assets/js/entities/repository.js

```
1  Entities.Repository = Backbone.Model.extend({
2    initialize: function(){
3      // edited for brevity
4    },
5
6    urlRoot: "https://api.github.com",
7
8    // edited for brevity
9
10   sync: function(method, model, options){
11     var username = "USERNAME",
12         password = "PASSWORD",
13         config = {
14           // edited for brevity
15         };
16
17     switch(method){
18       case "create":
19         config = _.extend(config, {
20           method: "POST",
21           url: _.result(this, "urlRoot") + "/user/repos",
22
```

```
23          // edited for brevity
24  });
```

We've added the `urlRoot` property on line 6, and removed the `baseUrl` variables declaration from line 13. Since `baseUrl` no longer exists, we need to remove all references to it, just like on line 21.

Since `urlRoot` can be defined as both a value and a function (see documentation[64]), we use Underscore's `result` function (documentation[65]) to give us the resulting value, regardless of `urlRoot`'s implementation: it takes the target object, and property name as arguments. Naturally, all other references to `baseUrl` in this file must be similarly replaced, but aren't displayed for conciseness.

Let's take a look at the various `url` properties we use when interacting with the API:

- create: `urlRoot` + "/user/repos"
- read: `urlRoot` + "/repos/" + username + "/" + model.get("name")
- update: `urlRoot` + "/repos/" + username + "/" + model.get("githubName")
- delete: `urlRoot` + "/repos/" + username + "/" + model.get("githubName")

`urlRoot` is displayed as pseudo-code for the `_.result()` call used above, for clarity.

There's a lot of similarity isn't there? Let's define the model's `url` property as the most common case, and overwrite it for the "create" case. Here we go:

assets/js/entities/repository.js

```
1  urlRoot: "https://api.github.com",
2  url: function(){
3    return this.urlRoot + "/repos/" + username + "/" +
4                     (model.get("githubName") || model.get("name"));
5  },
```

Note how we need to see if `githubName` exists (e.g. on update/delete calls), and fall back on the `name` attribute (for read calls) if it doesn't.

Except this won't work: the `username` and `model` values aren't in the scope. Let's fix that by setting `username` and `password` attributes when the model gets initialized, and using `this` to refer to the model instance:

[64] http://backbonejs.org/#Model-urlRoot
[65] http://underscorejs.org/#result

assets/js/entities/repository.js

```
Entities.Repository = Backbone.Model.extend({
  initialize: function(){
    this.username = "USERNAME";
    this.password = "PASSWORD";

    var self = this;
    this.on("sync", function(){
      self.set({githubName: self.get("name")}, {silent:true});
    });
  },

  urlRoot: "https://api.github.com",
  url: function(){
    return _.result(this, "urlRoot") + "/repos/" + this.username +
                      "/" + (this.get("githubName") || this.get("name"));
  },
```

Much better. And with this change in place, we can remove the `url` attributes for most API calls (they'll use the `url` function). But we'll still have to overwrite the `url` value for the "create" call, because it's different:

assets/js/entities/repository.js

```
Entities.Repository = Backbone.Model.extend({
  initialize: function(){
    this.username = "USERNAME";
    this.password = "PASSWORD";

    var self = this;
    this.on("sync", function(){
      self.set({githubName: self.get("name")}, {silent:true});
    });
  },

  urlRoot: "https://api.github.com",
  url: function(){
    return _.result(this, "urlRoot") + "/repos/" + this.username +
                      "/" + (this.get("githubName") || this.get("name"));
  },

  validate: function(attrs, options) {
```

```
19      // edited for brevity
20    },
21
22    sync: function(method, model, options){
23      var self = this,
24          config = {
25            beforeSend: function (xhr) {
26              xhr.setRequestHeader ("Authorization", "Basic " +
27                           btoa(self.username + ":" + self.password));
28            }
29          };
30
31      switch(method){
32        case "create":
33          config = _.extend(config, {
34            method: "POST",
35            url: _.result(this, "urlRoot") + "/user/repos",
36            data: JSON.stringify(model.pick("name", "description", "homepage",
37              "private", "has_issues", "has_wiki", "has_downloads", "team_id",
38              "auto_init", "gitignore_template"
39            ))
40          });
41          break;
42
43        case "read":
44          config = _.extend(config, {
45            method: "GET"
46          });
47          break;
48
49        case "update":
50          config = _.extend(config, {
51            method: "PATCH",
52            data: JSON.stringify(model.pick("name", "description", "homepage",
53              "private", "has_issues", "has_wiki", "has_downloads", "default_branc\
54  h"
55            ))
56          });
57          break;
58
59        case "delete":
60          config = _.extend(config, {
```

```
            method: "DELETE"
          });
          break;
      };

      // add API call configuration to the `options` object
      options = _.extend(options, config);

      return Backbone.Model.prototype.sync.call(this, method, model, options);
    }
});
```

If you're going to have a lot of repository instances, you'd be wise to move most of these functions into the Repository prototype to save memory (since they're common to all repository instances):

```
Entities.Repository = Backbone.Model.extend({
  // definitions/attributes that are unique to this model
  //
  // this will depend on your use case, but one example
  // could be username/password if
  // you're modifying the repositories for multiple users. For example:
  initialize(attributes, options){
    options || (options = {});
    this.username = options.username;
    this.password = options.password;
  }
});

_.extend(Entities.Repository.prototype, {
    // function definitions: initialize, sync, etc.
});

// You could then instanciate models requiring different credentials with
var repo = new ContactManager.Entities.Repository({name: "testo"},
                              {username: "myUsername", password: "myPassword"});
```

Redefining the `url` Property for Legacy APIs

We've seen how to define the model's `url` property, as well as how to call prototype methods. Let's say we have an API with the following endpoints:

- POST request to *repositories* to create a new repository;
- GET/PUT/DELETE request to *repositories/ID.json* to read/update/delete a repository.

As an exercise, implement the model's `url` function to return the proper value.

Since the *json* extension is added only for existing models, we can call the original implementation and then add the extension if necessary:

```
url: function(){
  var result = Backbone.Model.prototype.url.call(this);
  if( ! this.isNew()){
    result += ".json"
  }
  return result;
}
```

Git commit implementing the missing CRUD actions on GitHub's API:

b337b738442049254ac69927db463101fd08581f[66]

Dealing with API errors

Since we've properly implemented our API wrapping, we can deal with any errors as usual (i.e. using an `error` callback):

```
// create a repository named "repo" and save it
var repo = new ContactManager.Entities.Repository({name: "repo"});
repo.save();

// try creating a new repository with the same name
repo = new ContactManager.Entities.Repository({name: "repo"});
repo.save({}, {
  error: function(model, response, options){
    var message = "Unable to save repository '" + model.get("name") + "':\n";
    _.each(response.responseJSON.errors, function(e){
      message += "\t" + e.field + ": " + e.message;
    });
    console.log(message);
  }
});
```

As you've now learned, wrapping APIs in native Backbone models allows us to redefine only what is needed, while benefiting from the predefined default behavior (such as having our success and error callbacks handled automatically for us).

[66] https://github.com/davidsulc/marionette-serious-progression-app/commit/b337b738442049254ac69927db463101fd08581f

Managing Authentication

Intercepting Errors with `sync`

Some "generic" errors can be caught in a higher-level, centralized `sync` function to reduce code duplication. Let's see how we can intercept authentication errors without needing to duplicate the verification code everywhere. What we'd like to do is to have our app show a dialog anytime we're trying to do something that is not authorized, while calling the error callback if the error is not related to authorization.

This type of centralized error processing is useful to intercept session timeouts: if a user's session times out, the error can be intercepted and the user can be made to log in again (e.g. by redirecting him to the login page by changing `window.location`). This saves you from having to check for timed out sessions on each API call.

In this chapter, we'll continue using Github's API as in the previous chapter.

First, let's change our repository initialization code so we can provide different usernames and passwords:

assets/js/entities/repository.js

```
1  Entities.Repository = Backbone.Model.extend({
2    initialize: function(options){
3      options || (options = {});
4      this.username = options.username || "USERNAME";
5      this.password = options.password || "PASSWORD";
6
7      var self = this;
8      this.on("sync", function(){
9        self.set({githubName: self.get("name")}, {silent:true});
10     });
11   },
12
13   // edited for brevity
```

Let's trigger an unauthorized API call from the console to verify the error (leave the `username` and `password` values as they are below):

Console input

```
var r = new ContactManager.Entities.Repository({
  username: "XXX",
  password: "YYY",
  name:"my-repo"
});
r.fetch();
```

You can see we get a "401 Unauthorized" response from the API. Let's try and process that error in the `fetch` call:

Console input

```
var r = new ContactManager.Entities.Repository({
  username: "XXX",
  password: "YYY",
  name:"my-repo"
});
r.fetch({
  error: function(model, response, options){
    if(response.status === 401){
      alert("This action isn't authorized!");
    }
  }
});
```

GitHub's API will prevent you from making too many failed authentication attempts, for security reasons: it will return a "403 Forbidden" status once too many attempts have been made. If this happens, just wait a while before trying again. Also, be a good netizen and don't flood their servers with requests!

Great, we're able to display the alert if the action isn't authorized by the API. Let's add to our error management to make sure the alert doesn't get displayed for all errors (this time, replace the `username` and `password` with yours):

Console input

```
var r = new ContactManager.Entities.Repository({
  username: "USERNAME",
  password: "PASSWORD",
  name:"my-repo"
});
r.fetch({
  error: function(model, response, options){
    if(response.status === 401){
      alert("This action isn't authorized!");
    }
    else{
      console.log("This action is authorized but still has an error: ",
                  response.responseJSON.message);
    }
  }
});
```

This time around, we can see we've got a "not found" error, so we're differentiating errors properly. Now, we need to manage this at a higher level, by rewriting `Backbone.sync`. The main idea is to call the original `Backbone.sync` function, and then process the jQuery promise[67] value it returns.

Here we go[68]:

assets/js/entities/common.js

```
Entities.BaseModel = Backbone.Model.extend({
  // edited for brevity
});

var originalSync = Backbone.sync;
Backbone.sync = function (method, model, options) {
  var deferred = $.Deferred();
  options || (options = {});
  deferred.then(options.success, options.error);

  var response = originalSync(method, model,
                              _.omit(options, 'success', 'error'));

```

[67] http://api.jquery.com/Types/#Promise

[68] This rewrite implementation was adapted from nikoshr's answer at http://stackoverflow.com/questions/16476874/catching-backbone-sync-errors

```
14    response.done(deferred.resolve);
15    response.fail(function() {
16      if(response.status == 401){
17        alert("This action isn't authorized!");
18      }
19      else if(response.status === 403){
20        alert(response.responseJSON.message);
21      }
22      else{
23        deferred.rejectWith(response, arguments);
24      }
25    });
26
27    return deferred.promise();
28  };
```

First, we keep a reference to the original `Backbone.sync` implementation on line 5. Note that in some cases, you might want to add this reference to the `Backbone` object itself, which is what the `localstorage` plugin does: it registers the original `sync` implementation as `Backbone.ajaxSync`.

On line 7, we create a new jQuery Deferred[69] object instance which we'll then use to process the `sync` result and execute the appropriate callbacks. Line 8 makes sure we have a valid `options` object, either by using the one provided or by assigning an empty object: we can then be sure that accessing the `success` and `error` attributes won't raise an error (although they migh be undefined). Then (line 9), if any `options` were provided, we register them as the success/error callbacks to run when our deferred object is marked as resolved (i.e. the asynchronous `sync` function has returned).

We call the original `sync` implementation on line 11, which returns a promise[70] we can monitor and react to. In fact, we immediately add callbacks so we can react accordingly:

- the `done`[71] call on line 14 registers the callback to execute if the reponse is successful. In that case, we simply resolve our deferred object, which will in turn execute our "success" callback;
- the `fail`[72] call on lines 15-25 registers the callback to execute if an error is returned in the response. We display a message if the status is 401 (unauthorized) or 403 (forbidden, which will happen if there are too many failed login attempts) on lines 16-21. But on lines 22-24, we use `rejectWith`[73] to reject our deferred object by providing the original server response as the context, followed by the arguments our `fail` callback was called with. This, in turn, executes the "error" callback.

[69] http://api.jquery.com/Types/#Deferred
[70] http://api.jquery.com/Types/#Promise
[71] http://api.jquery.com/deferred.done/
[72] http://api.jquery.com/deferred.fail/
[73] http://api.jquery.com/deferred.rejectWith/

Note that on line 23 we could have achieved the same result by using

```
deferred.reject.apply(response, arguments);
```

Finally, we return the promise associated to our deferred object (line 27), so any function can register additional callbacks (e.g. to run when the `sync` call is done).

In our case all we wanted to do if we weren't authorized was to display an alert, we didn't want to execute the original `error` callback. Not calling the original error callback is what you would do (e.g.) if you're handling an error due to a timed out session: you'd only want to redirect to the login page, not run the error callback. If you actually *do* need to execute the original error callback, simply insert line 23 where appropriate.

Let's try our new `Backbone.sync` code:

Console input

```
1  // unauthorized access
2  var r = new ContactManager.Entities.Repository({
3    username: "XXX",
4    password: "YYY",
5    name:"my-repo"
6  });
7  r.fetch({
8    error: function(model, response, options){
9        console.log("This action is authorized but still has an error: ",
10                   response.responseJSON.message);
11   }
12 });
13
14 // authorized access (replace with your username/password)
15 var r = new ContactManager.Entities.Repository({
16   username: "USERNAME",
17   password: "PASSWORD",
18   name:"my-repo"
19 });
20 r.fetch({
21   error: function(model, response, options){
22       console.log("This action is authorized but still has an error: ",
23                   response.responseJSON.message);
24   }
```

```
});

// success callback works as expected (make sure "my-repository" exists!)
var r = new ContactManager.Entities.Repository({
  username: "USERNAME",
  password: "PASSWORD",
  name:"my-repository"
});
r.fetch({
  success: function(){
    console.log("Successfully fetched repository!");
  },
  error: function(model, response, options){
      console.log("This action is authorized but still has an error: ",
                  response.responseJSON.message);
  }
});
```

Git commit intercepting authentication errors with `sync`:

b4b7c874d50f330f4c41bb7ab31cadfab6162310[74]

Now that we've covered rewriting `Backbone.sync`, you have the means to intercept and process errors that are applicable to all Backbone API calls. You can also register a general error callback with jQuery ajaxError[75], but it will be executed for *all* ajax requests that fail (not just those initiated by Backbone), so it might be too broad-scoped, depending on your use case.

Other Authentication Possibilities

In the GitHub example we've gone through above, authentication was handled using HTTP basic authentication, providing the username/password information on each request.

Another, more common way to handle authentication is with sessions. This is extremely common with web apps, as they are often accessed after logging in (and therefore with an active session). To handle authentication with sessions, you simply need an end point on your API that will create an authenticated session when receiving a valid username/password combination. That's it!

Once the session is created on the server (check the documentation for your favorite framework on how to do this), your client will typically have a cookie containing the session information. And

[74] https://github.com/davidsulc/marionette-serious-progression-app/commit/b4b7c874d50f330f4c41bb7ab31cadfab6162310

[75] http://api.jquery.com/ajaxError/

since this cookie gets sent with each request to the server, it will be able to differentiate authenticated requests.

The only thing left to handle is session timeouts. These should be managed as above, in the `sync` function: on every error, if the error is due to a timed out session (per the status code), redirect the user to the login page.

Managing Authorization on the Client Side

With users authenticated, you'll typically have different templates sent to the client according to access rights (but once again: don't trust the client, verify all access on the server). But, for educationnal purposes, here's an object you could use to manage that access on the client:

```
var User = function(config){
  this.roles = config.roles || [];
  this.rights = config.rights || [];
};

_.extend(User.prototype, {
  can: function(right, klass){
    return this.hasRole("admin") || this.hasGenericRight(right) ||
                                    this.hasSpecificRight(right, klass);
  },

  hasRole: function(role){
    console.log(this.roles);
    return this.roles.indexOf(role) > -1;
  },

  hasGenericRight: function(right){
    var genericRights = this.rights["all"];
    return genericRights && genericRights.indexOf(right) > -1;
  },

  hasSpecificRight: function(right, klass){
    var objectRights = this.rights[klass];
    if(! objectRights){ return false; }
    return objectRights.indexOf(right) > -1;
  }
});
```

It uses multiple levels to determine what a user can do. It checks whether the user has an "admin" role, whether he has a generic access right (e.g. "read everything"), or a specific access right for

certain object types (e.g. “edit contacts”).

You can then initialize this object based on the access information provided by the server. For example:

```
var user = new User({ rights: { all: ["read"], contact: ["edit"] } });
user.can("read", "contact"); // true
user.can("edit", "contact"); // true
user.can("delete", "contact"); // false
```

Based on the boolean value of the `user.can` calls, you can determine whether (e.g.) a button should be displayed on the interface or not.

Handling Server-Side Responses

API Properties

The "contacts" endpoint will return a 404 "not found" error if no contact with the given id is found. In addition, it implements the following validations on the server side:

- first name: can't be blank
- last name: can't be blank, must be at least 2 characters long
- phone number: must be unique (i.e. 2 separate contacts cannot have the same string as a phone number)

If server-side validations fails, a 422 "unprocessable entity" error will be returned and the payload will contain an `entity` key containing the contact entity in its currently persisted state (on the server), along with an `errors` key associating each incorrect attribute with an array of error messages:

```
{
  entity: {
    id: 1,
    firstName: "Alice",
    lastName: "Arten",
    phoneNumber: "555-0184"
  },

  errors: {
    firstName:   ["can't be blank"],
    lastName:    ["is too short (minimum is 2 characters)"],
    phoneNumber: ["has already been taken"]
  }
}
```

Processing 404 Errors

Let's take a look at the code we use to fetch a contact:

assets/js/entities/contact.js

```
getContactEntity: function(contactId){
  var contact = new Entities.Contact({id: contactId});
  var defer = $.Deferred();
  contact.fetch({
    success: function(data){
      defer.resolve(data);
    },
    error: function(data){
      defer.resolve(undefined);
    }
  });
  return defer.promise();
}
```

It works fine for our current use: return the contact if it's found, and if there's an error we return the `undefined` value. But with this code, there's no way for us to differentiate a 404 error (Not Found) from (e.g.) a 500 error (Internal Error). Instead, we should be able to check the status code and respond accordingly. To accomplish this, we'll use our trusty deferreds much like in our `Backbone.sync` reimplementation (see here):

assets/js/entities/contact.js

```
1  getContactEntity: function(contactId, options){
2    var contact = new Entities.Contact({id: contactId});
3    var defer = $.Deferred();
4    options || (options = {});
5    defer.then(options.success, options.error);
6    var response = contact.fetch(_.omit(options, 'success', 'error'));
7    response.done(function(){
8      defer.resolveWith(response, [contact]);
9    });
10   response.fail(function(){
11     defer.rejectWith(response, arguments);
12   });
13   return defer.promise();
14 }
```

Let's start by looking at what stayed the same: on line 3 we create a deferred object, and we then return its promise on line 13. Then, on lines 4-5, we make sure the `options` object exists and use

its `success` and `error` attributes to define the callbacks that should be executed when `defer` is completed.

On line 6, we execute the actual `fetch` call, and store a reference to the response (which is in fact a deferred object itself). If the `fetch` executes successfully, we resolve our deferred object (`defer`) with `response` as the context (because that's the actual response that came from the server), and since `resolveWith` takes an array as a second argument, we provide an array containing only our fetched `contact` (lines 7-9).

On lines 10-12, we reject our deferred with the server `response` as the context, and pass along the arguments provided to the `fail` call. Passing the arguments on will allow any objects listening to the promise to be able to access them. As we'll see below, this will be handy to determine the response status in the controller.

The `resolveWith` and `rejectWith` methods are simply syntactic sugar, and could also be implemented as

```
defer.resolve.call(response, contact);
```

and

```
defer.reject.apply(response, arguments);
```

respectively.

Of course, now that our `getContactEntity` function takes an `options` argument, we need to pass it on in our handler:

assets/js/entities/contact.js

```
1 ContactManager.reqres.setHandler("contact:entity", function(id, options){
2   return API.getContactEntity(id, options);
3 });
```

Reacting to Server Response Codes

With our contact fetching properly passing on status codes, let's manage them in the controller code. Let's take a look at our current implementation:

assets/js/apps/contacts/show/show_controller.js

```
1  var fetchingContact = ContactManager.request("contact:entity", id);
2  $.when(fetchingContact).done(function(contact){
3    var contactView;
4    if(contact !== undefined){
5      contactView = new Show.Contact({
6        model: contact
7      });
8
9      contactView.on("contact:edit", function(contact){
10       ContactManager.trigger("contact:edit", contact.get("id"));
11     });
12   }
13   else{
14     contactView = new Show.MissingContact();
15   }
16
17   ContactManager.mainRegion.show(contactView);
18 });
```

As you can see, we attempt to fetch the contact on line 1, then check the return value on line 4. Since we're now making better use of the deferred, let's implement a `fail` callback:

assets/js/apps/contacts/show/show_controller.js

```
1  var fetchingContact = ContactManager.request("contact:entity", id);
2  $.when(fetchingContact).done(function(contact){
3    var contactView = new Show.Contact({
4      model: contact
5    });
6
7    contactView.on("contact:edit", function(contact){
8      ContactManager.trigger("contact:edit", contact.get("id"));
9    });
10
11   ContactManager.mainRegion.show(contactView);
12 }).fail(function(response){
13   console.log("Some error happened (processed in deferred's fail callback)");
14   if(response.status === 404){
15     var contactView = new Show.MissingContact();
16     ContactManager.mainRegion.show(contactView);
17   }
```

```
18    else{
19      alert("An unprocessed error happened. Please try again!");
20    }
21  });
```

What's going on here? At the higher level, you can see there are essentially 3 code sections:

1. On lines 1-2 we keep a reference to the deferred object, and wait for the server response (using `$.when()`);
2. Lines 2-12 provide a callback function to use if the deferred request on line 1 is successful;
3. Lines 12-21 provide a callback function to use if the deferred request on line 1 fails.

In the above code, we chain the callback definitions but we could just as well have defined them separately:

```
1  var fetchingContact = ContactManager.request("contact:entity", id);
2  $.when(fetchingContact).done(...);
3  // some unrelated code could go here
4  $.when(fetchingContact).fail(...);
```

The success callback simply executes the same code as before, while the error callback reacts differently by checking the response status: if the contact wasn't found, it displays the "missing contact" view. However, if the request failed for another reason (e.g. an internal server error), an alert is displayed instead.

The code above to display a contact is fully functional and behaves as expected. But if you'll recall, we implemented the `request` function to be able to use success and error callbacks provided in an options object. Let's use those, too:

assets/js/apps/contacts/show/show_controller.js

```
1  var fetchingContact = ContactManager.request("contact:entity", id, {
2    error: function(xhr, responseText, error){
3      console.log("Some error happened (processed in error callback)");
4    }
5  });
6  $.when(fetchingContact).done(function(contact){
7    // edited for brevity
8  }).fail(function(response){
9    // edited for brevity
10 });
```

If we now try to load an invalid contact id (e.g. by entering "#contacts/test" in the address bar), we'll see that both of our error messages get displayed in the console. Excellent!

Adapting the Edit View

As you've most likely guessed, we also need to correct our "edit" code to properly react to server status codes when the contact is being fetched. Try implementing that on you own, then continue reading for the solution.

There really isn't much going on here:

assets/js/apps/contacts/edit/edit_controller.js

```
var fetchingContact = ContactManager.request("contact:entity", id);
$.when(fetchingContact).done(function(contact){
  var view = new Edit.Contact({
    model: contact,
    generateTitle: true
  });

  view.on("form:submit", function(data){
    if(contact.save(data)){
      ContactManager.trigger("contact:show", contact.get("id"));
    }
    else{
      view.triggerMethod("form:data:invalid", contact.validationError);
    }
  });

  ContactManager.mainRegion.show(view);
}).fail(function(response){
  if(response.status === 404){
    var view = new ContactManager.ContactsApp.Show.MissingContact();
    ContactManager.mainRegion.show(view);
  }
  else{
    alert("An unprocessed error happened. Please try again!");
  }
});
```

Reacting to Server-Side Validation Errors

We're now reacting to the provided status code when we fetch the contact to edit. But we also need to respond to the status codes when the edited contact is *saved*. In order to trigger the server-side errors (so that they aren't first caught by server-side validations), let's comment the client-side validation code so that incorrect data gets sent to the server and we can react to the server response:

assets/js/entities/contact.js

```
validate: function(attrs, options) {
  //var errors = {}
  //if (! attrs.firstName) {
  //  errors.firstName = "can't be blank";
  //}
  //if (! attrs.lastName) {
  //  errors.lastName = "can't be blank";
  //}
  //else{
  //  if (attrs.lastName.length < 2) {
  //    errors.lastName = "is too short";
  //  }
  //}
  //if( ! _.isEmpty(errors)){
  //  return errors;
  //}
},
```

If we now go to the edit page for a contact (at *#contacts/ID/edit*), edit the contact to have a blank last name, then click "Update contact", the following will take place:

1. The API will return an error (422 Unprocessable Entity);
2. The user will be redirected to the contact's "show" view with unaltered data.

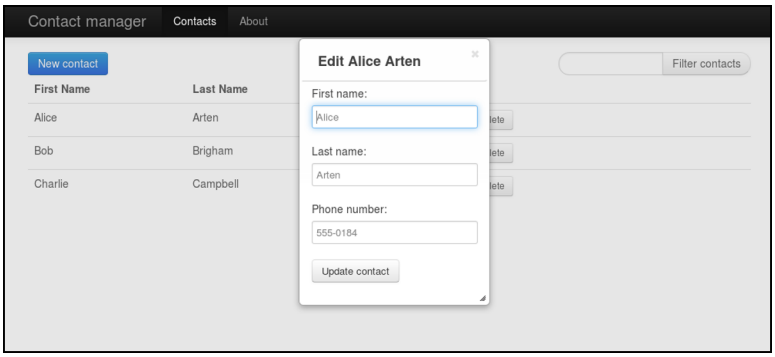
Let's take a look at our current "edit" code:

assets/js/apps/contacts/edit/edit_controller.js

```
var fetchingContact = ContactManager.request("contact:entity", id);
$.when(fetchingContact).done(function(contact){
  var view = new Edit.Contact({
    model: contact,
    generateTitle: true
  });

  view.on("form:submit", function(data){
    if(contact.save(data)){
      ContactManager.trigger("contact:show", contact.get("id"));
    }
    else{
```

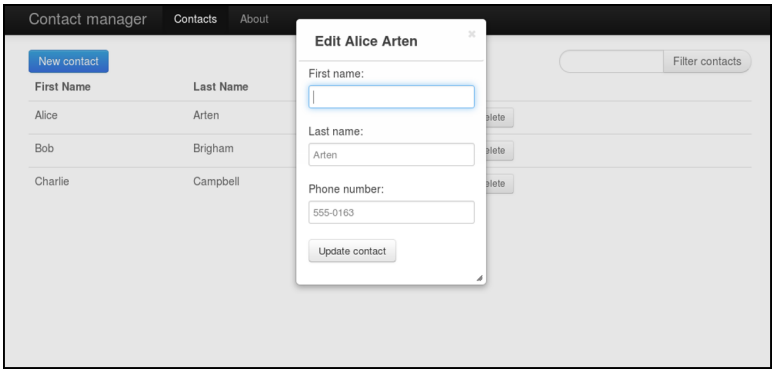
```
13        view.triggerMethod("form:data:invalid", contact.validationError);
14      }
15    });
16
17    ContactManager.mainRegion.show(view);
18  }).fail(function(response){
19    if(response.status === 404){
20      var view = new ContactManager.ContactsApp.Show.MissingContact();
21      ContactManager.mainRegion.show(view);
22    }
23    else{
24      alert("An unprocessed error happened. Please try again!");
25    }
26  });
```

Lines 8-15 are of particular interest to us, since they're the root of our problem. On line 9, we simply check the return value of the `save` call. If you take a look at the documentation[76], you'll see that it returns `false` if the client-side validation fails (as implemented by the model's `validate` method), and returns the deferred value otherwise. Since the client-side validation passes (because it has been commented), `save` will return a truthy value and we'll end up navigating to the contact's "show" view. Let's visualize the process:

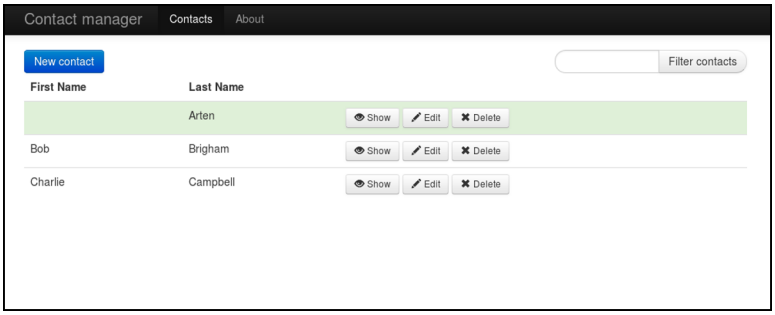


Editing a contact

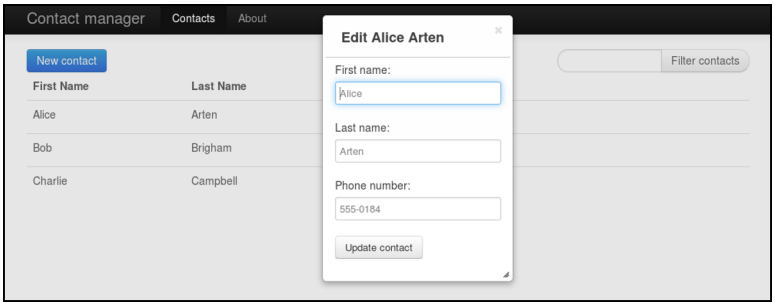
[76] http://backbonejs.org/#Model-save



Entering invalid data (empty first name and duplicate phone number)



Saving the invalid data (notice the blank first name displayed)



Refreshing the page and reopening the modal "edit" window

If all we were interested in was waiting for the API response before doing anything, we could simply add the `wait` option in the code above:

```
if(contact.save(data, {wait: true})){
```

but this doesn't solve our problem since then we'll just be redisplaying an unmodified contact: validation errors still won't be displayed.

What we'd want to have happen instead is for the server-side errors to be displayed on our form. Let's start by using the `save` deferred:

assets/js/apps/contacts/edit/edit_controller.js

```
view.on("form:submit", function(data){
  var savingContact = contact.save(data);
  $.when(savingContact).done(function(){
    ContactManager.trigger("contact:show", contact.get("id"));
  }).fail(function(response){
    console.log("There has been an error: ", response.responseJSON);
    view.triggerMethod("form:data:invalid", contact.validationError);
  });
});
```

Pretty simple: if saving the contact works without errors, we proceed as usual and trigger the "contact:show" event. If it fails, we instead display the response content to the console. Now, if you go to a contact's edit page and erase the first and last names, you can see what the server response's `responseJSON` object looks like:

```
{
  firstName: ["can't be blank"],
  lastName: ["is too short (minimum is 2 characters)"]
}
```

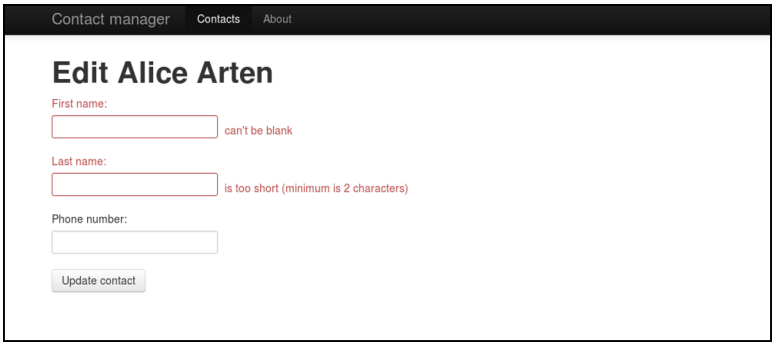
In other words, the API returns an object containing attributes that caused errors (as the object's keys), as well as a list of problem descriptions. So what we need to do is check if the error is due to validation problems, and if so, add the errors to the contact's `validationError` object:

assets/js/apps/contacts/edit/edit_controller.js

```
view.on("form:submit", function(data){
  var savingContact = contact.save(data);
  $.when(savingContact).done(function(){
    ContactManager.trigger("contact:show", contact.get("id"));
  }).fail(function(response){
    if(response.status === 422){
      view.triggerMethod("form:data:invalid", response.responseJSON.errors);
    }
    else{
      alert("An unprocessed error happened. Please try again!");
    }
  });
});
```

Our API will return one error per attribute, at most. If this isn't the case for you, you'll need to properly process and display all error messages.

If we now try saving a contact with an empty first and last name, we get the following:



Displaying server-side validation errors

Let's restore the validation code:

assets/js/entities/contact.js

```
validate: function(attrs, options) {
  var errors = {}
  if (! attrs.firstName) {
    errors.firstName = "can't be blank";
  }
  if (! attrs.lastName) {
    errors.lastName = "can't be blank";
  }
  else{
    if (attrs.lastName.length < 2) {
      errors.lastName = "is too short";
    }
  }
  if( ! _.isEmpty(errors)){
    return errors;
  }
},
```

When we try saving an invalid contact again (e.g. with an empty first name), we're immediately redirected to the "show" page, and an API request doesn't even get fired. Why is that? Let's look at a simplified version of our "edit" code:

assets/js/apps/contacts/edit/edit_controller.js

```
view.on("form:submit", function(data){
  var savingContact = contact.save(data);
  $.when(savingContact).done(function(){
    ContactManager.trigger("contact:show", contact.get("id"));
  }).fail(function(response){
    // edited for brevity
  });
});
```

Here's what happens when we try to save an invalid contact: as we said earlier, `save` will return `false` if the model fails validation. This `false` value is then provided as the argument to `jQuery.when`, which according to the documentation[77] behaves like this:

> If a single argument is passed to jQuery.when and it is not a Deferred or a Promise, it will be treated as a resolved Deferred and any doneCallbacks attached will be executed immediately.

So in our case, the `save` call is considered immediately successful and the `done` callback is executed, displaying the "show" page. To address this issue, we simply have to verify the `save` call's return value:

assets/js/apps/contacts/edit/edit_controller.js

```
view.on("form:submit", function(data){
  var savingContact = contact.save(data);
  if(savingContact){
    $.when(savingContact).done(function(){
      ContactManager.trigger("contact:show", contact.get("id"));
    }).fail(function(response){
      if(response.status === 422){
        view.triggerMethod("form:data:invalid", response.responseJSON.errors);
      }
      else{
        alert("An unprocessed error happened. Please try again!");
      }
    });
  }
  else{
```

[77] http://api.jquery.com/jQuery.when/

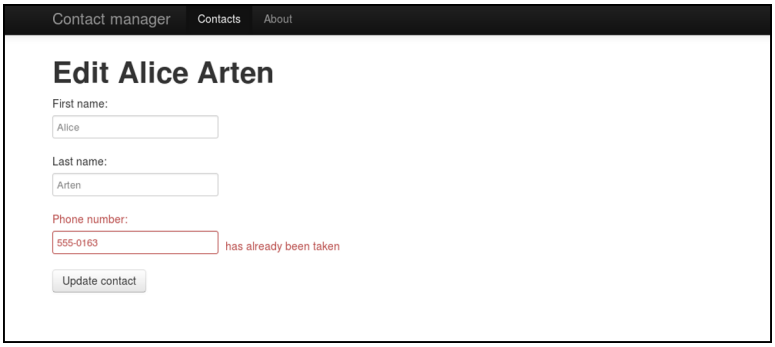
```
16      view.triggerMethod("form:data:invalid", contact.validationError);
17    }
18  });
```

This could also have been implemented using callbacks, which we'll explain in the next section.

Git commit managing the server response when the contact is edited from the dedicated page

5b265ce6721e9e51ce6edeae35130e6d290a0cac[78]

Now the client-side validation works as expected: an API request is only fired if the client-side validations pass. Then, the data is checked again on the server side, and if it is unsuccessful the error will be returned and displayed on the form. You can see this in action for yourself by attempting to save the edited contact by using a phone number that already belongs to another contact (in other words, the API enforces phone number uniqueness among contacts):



Note that *all* validation **MUST** be enforced by the API (i.e. server-side) for security reasons: client-side validations can be tampered with, client-side data could be out of date leading uniqueness validations to pass/fail when they shouldn't, etc.

Deferreds Versus Success/Error Callbacks

The previous code modification could also have been accomplished using callbacks:

[78] https://github.com/davidsulc/marionette-serious-progression-5b265ce6721e9e51ce6edeae35130e6d290a0cac

assets/js/apps/contacts/edit/edit_controller.js

```
view.on("form:submit", function(data){
  var savingContact = contact.save(data, {
    sucess: function(){
      ContactManager.trigger("contact:show", contact.get("id"));
    },
    error: function(model, response, options){
      if(response.status === 422){
        view.triggerMethod("form:data:invalid", response.responseJSON.errors);
      }
      else{
        alert("An unprocessed error happened. Please try again!");
      }
    }
  });
  if( ! savingContact){
    view.triggerMethod("form:data:invalid", contact.validationError);
  }
});
```

So why did we use deferreds instead? Mainly personal preference. I find deferreds are more flexible, and therefore tend to prefer them: they allow for code synchronization, deferred callback (e.g. `done` and `fail` callbacks) can be defined once the deferred has already returned (they will be executed immediately), and more. In addition, since we can chain the `done` and `fail` callback definitions, providing success/error callbacks doesn't offer better readability. Besides, don't forget: as mentioned above, you can still structure your code to use both success/error callbacks *and* deferreds, if that's what you prefer!

Validations: Client-Side Versus Server-Side

If all validations need to be enforced on the server, why bother having client-side validation at all? It's there mainly for the user experience: we don't need to wait for the server response to (e.g.) tell the user that the contact's last name is too short. Having the "atomic" validation (i.e. independent from other data) on the client makes for a more reactive user experience.

Why haven't we implemented the phone number uniqueness constraint on the client side? Because it would be more work than it would be worth: we'll rarely have a significant amount of the data available on the client, so we'd just be adding complexity to our application for questionable gains.

The direct consequence of this two-phased validation is that users may see errors displayed in 2 phases. It sounds silly when I say it like that, but if a user edits a contact with a missing last name and a duplicate phone number, the following will happen:

1. The user edits the contact and submits the form;
2. An error indicates that the last name can't be blank (client-side validation);
3. The user enters a last name and submits the form again;
4. Another error indicates that the phone number has already been taken (server-side validation);
5. The user needs to correct the data again, even though the phone number "passed" the first validation.

In many cases, this isn't an issue because it usually saves time for the user: errors that can be checked locally get immediate feedback and can be fixed without requiring a server round-trip. The paradigm I typically follow is:

- data that can be verified "atomically" (i.e. regardless of what might exist on the server or elsewhere in the system) is validated with client-side validation;
- data that is context-sensitive (e.g. phone number uniqueness) is validated only on the server;
- *everything* gets validated on the server, including the client-side validations, for security purposes.

Of course, the above might not suit your purposes or planned user interactions. If you want to have all errors displayed at once, your best bet is to ignore client-side validations and use server-side validation exclusively (at the expense of reactivity).

Fixing the Modal Edit

We're now correctly managing server responses when a contact is edited from the dedicated page (i.e. a URL like *#contacts/ID/edit*), but if you click on the "edit" button from the main page (*#contacts*) and we enter a duplicated phone number, the following happens:

- an API error is returned due to a failed server-side validation;
- it looks like the contact gets updated anyway: the line flashes green, and if you click on the "edit" button again, you'll see the duplicate phone number displayed.

However, if you refresh the page or click on the "show" button (both of which force data to be reloaded from the server), you will *not* see the duplicate phone number. In other words, the client-side model was updated with incorrect data, but it wasn't persisted on the server.

Our problem is that although client-side validation failures are handled correctly, server-side validations are disregarded… Here's the code responsible for this behavior:

assets/js/apps/contacts/list/list_controller.js

```
contactsListView.on("childview:contact:edit", function(childView, args){
  var model = args.model;
  var view = new ContactManager.ContactsApp.Edit.Contact({
    model: model
  });

  view.on("form:submit", function(data){
    if(model.save(data)){
      childView.render();
      view.trigger("dialog:close");
      childView.flash("success");
    }
    else{
      view.triggerMethod("form:data:invalid", model.validationError);
    }
  });

  ContactManager.dialogRegion.show(view);
});
```

Try and fix this problem, and check on the next page for the answer.

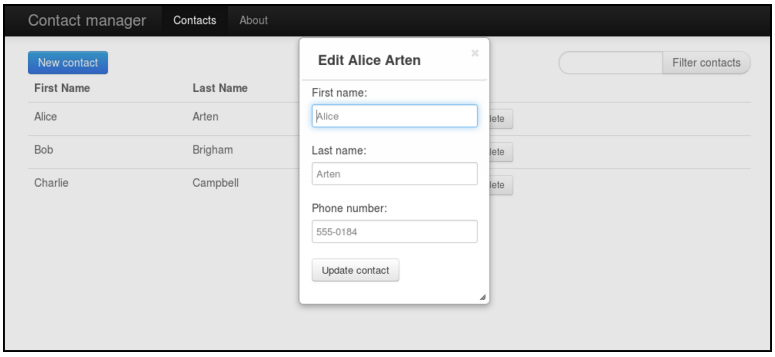
The solution is very similar to what we've developed for the "edit" action:

assets/js/apps/contacts/list/list_controller.js

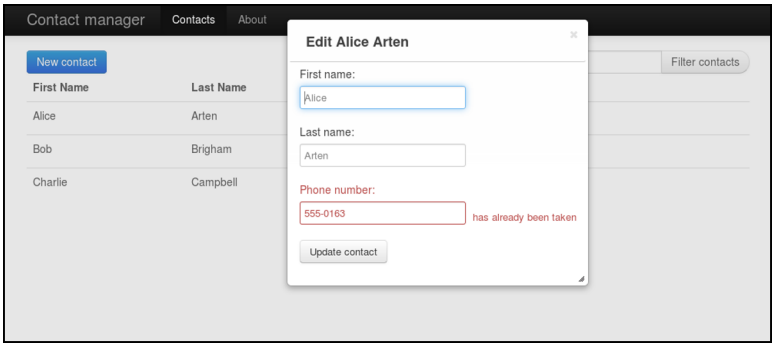
```
contactsListView.on("childview:contact:edit", function(childView, args){
  var model = args.model;
  var view = new ContactManager.ContactsApp.Edit.Contact({
    model: model
  });

  view.on("form:submit", function(data){
    var savingContact = model.save(data);
    if(savingContact){
      $.when(savingContact).done(function(){
        childView.render();
        view.trigger("dialog:close");
        childView.flash("success");
      }).fail(function(response){
        if(response.status === 422){
          view.triggerMethod("form:data:invalid",
                             response.responseJSON.errors);
        }
        else{
          alert("An unprocessed error happened. Please try again!");
        }
      });
    }
    else{
      view.triggerMethod("form:data:invalid", model.validationError);
    }
  });

  ContactManager.dialogRegion.show(view);
});
```

Restoring Valid Model Attributes as Required

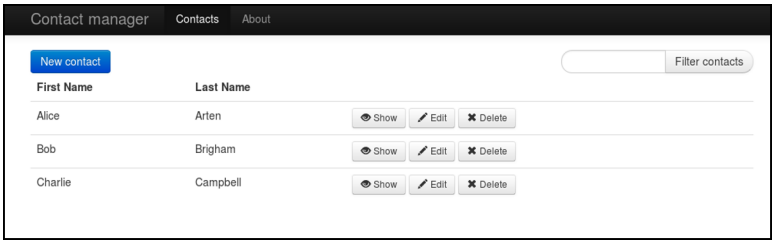
There's a bug in our app we need to fix: if we're unable to update the contact due to server-side validation errors, the local model instance still gets updated (although it isn't persisted). Take a look:



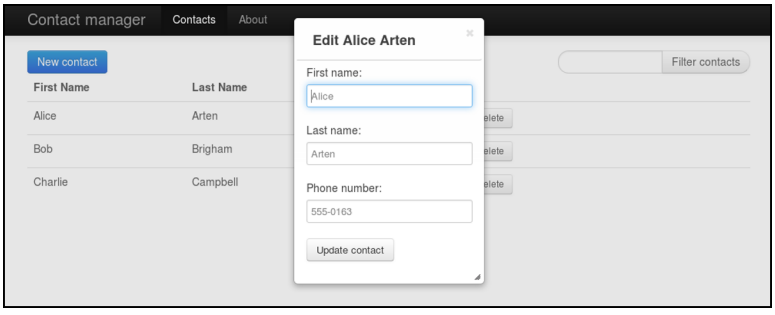
Editing a contact in a modal window



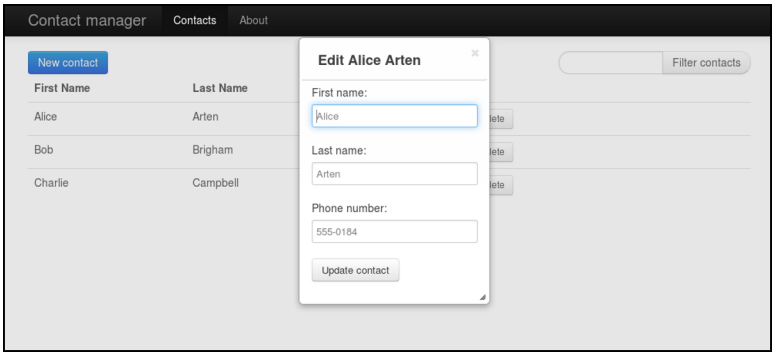
Attempting to save an invalid (duplicate) phone number



Closing the modal



Reopening the modal (notice the invalid phone number is displayed)



Refreshing the page and reopening the modal "edit" window (the phone number is fixed)

Try and debug this on your own, and take a look at the solution on the next page.

The solution to this bug is pretty straightforward: if the `save` call fails when we're editing a contact, we need to restore the model's previous attributes (since we know they were valid at the time).

assets/js/apps/contacts/list/list_controller.js

```
1  contactsListView.on("childview:contact:edit", function(childView, args){
2    // edited for brevity
3
4    view.on("form:submit", function(data){
5      var savingContact = model.save(data);
6      if(savingContact){
7        $.when(savingContact).done(function(){
8          // edited for brevity
9        }).fail(function(response){
10         view.onDestroy = function(){
11           model.set(model.previousAttributes());
12         };
13
14         if(response.status === 422){
15           // edited for brevity
```

As you can see on lines 10-12, we're restoring the model's attributes to their previous values in the `fail` callback.

Fixing the "New" Action

Right now, the code to create a new contact looks like this:

assets/js/apps/contact/list/list_controller.js

```
1  contactsListPanel.on("contact:new", function(){
2    var newContact = new ContactManager.Entities.Contact();
3
4    var view = new ContactManager.ContactsApp.New.Contact({
5      model: newContact
6    });
7
8    view.on("form:submit", function(data){
9      var contactSaved = newContact.save(data, {
10       success: function(){
11         contacts.add(newContact);
12         view.trigger("dialog:close");
```

```
13          var newContactView = contactsListView.children.findByModel(newContact);
14          // check whether the new contact view is displayed (it could be
15          // invisible due to the current filter criterion)
16          if(newContactView){
17            newContactView.flash("success");
18          }
19        }
20      });
21      if( ! contactSaved){
22        view.triggerMethod("form:data:invalid", newContact.validationError);
23      }
24    });
25
26    ContactManager.dialogRegion.show(view);
27  });
```

If we try creating a new contact with a duplicate phone number (which is forbidden by the server-side validation), an API error is returned but the user doesn't know about it: the modal window doesn't change or close. Nothing happens, and the app seems unresponsive!

Fix the code so that when a contact is created, both client- and server-side validations are processed properly, then check the next page for the solution.

Once again, we simply apply the same techniques as when addressing the "edit" case:

assets/js/apps/contacts/list/list_controller.js

```
contactsListPanel.on("contact:new", function(){
  var newContact = new ContactManager.Entities.Contact();

  var view = new ContactManager.ContactsApp.New.Contact({
    model: newContact
  });

  view.on("form:submit", function(data){
    var savingContact = newContact.save(data);
    if(savingContact){
      $.when(savingContact).done(function(){
        contacts.add(newContact);
        view.trigger("dialog:close");
        var newContactView = contactsListView.children.findByModel(newContact);
        // check whether the new contact view is displayed (it could be
        // invisible due to the current filter criterion)
        if(newContactView){
          newContactView.flash("success");
        }
      }).fail(function(response){
        view.onDestroy = function(){
          newContact.set(newContact.previousAttributes());
        };

        if(response.status === 422){
          view.triggerMethod("form:data:invalid",
                             response.responseJSON.errors);
        }
        else{
          alert("An unprocessed error happened. Please try again!");
        }
      });
    }
    else{
      view.triggerMethod("form:data:invalid", newContact.validationError);
    }
  });

  ContactManager.dialogRegion.show(view);
});
```

In this case, however, there's no need to restore the original model attributes when the dialog is closed: there was no existing contact instance before we tried creating a new one (so there are no previous attributes to revert to), and the modal view is always intantiated with a new contact instance.

Git commit managing server responses in the "new" and "edit" modals

c333df33c1d74e41b3c05b34af1e414d5e938907[79]

Handling Errors on Collection Fetch

Try and change the code so that if errors are thrown when the contacts collection is fetched, a generic message will be displayed for the user. Go on to the next page to see the updated code.

Note that our API shouldn't return an error, so any error returned would be due to some technical failure (e.g. "500 Internal error").

[79] https://github.com/davidsulc/marionette-serious-progression-c333df33c1d74e41b3c05b34af1e414d5e938907

First, we update our collection fetching code, just like we did when fetching a single contact:

assets/js/

```
var API = {
  getContactEntities: function(options){
    var contacts = new Entities.ContactCollection();
    var defer = $.Deferred();
    options || (options = {});
    defer.then(options.success, options.error);
    var response = contacts.fetch(_.omit(options, 'success', 'error'));
    response.done(function(){
      defer.resolveWith(response, [contacts]);
    });
    response.fail(function(){
      defer.rejectWith(response, arguments);
    });
    return defer.promise();
  },

  // edited for brevity
};

ContactManager.reqres.setHandler("contact:entities", function(options){
  return API.getContactEntities(options);
});

// edited for brevity
```

With that in place, we simply need to update our controller code to register a `fail` callback on the deferred.

assets/js/apps/contacts/list/list_controller.js

```
var fetchingContacts = ContactManager.request("contact:entities");

var contactsListLayout = new List.Layout();
var contactsListPanel = new List.Panel();

$.when(fetchingContacts).done(function(contacts){
  // edited for brevity
}).fail(function(){
  alert("An unprocessed error happened. Please try again!");
});
```

On line 1, we could also have specified some callbacks to execute when our fetch is done, for example

```
1 var fetchingContacts = ContactManager.request("contact:entities", {
2   success: function(){
3     console.log("fetch success!");
4   }
5 });
```

Git commit handling contact collection fetch failure with deferreds 25004bdfefb1c93b1b1b33115594215054720a1d[80]

[80] https://github.com/davidsulc/marionette-serious-progression-25004bdfefb1c93b1b1b33115594215054720a1d

Refreshing Client-Side Data

Beware Race Conditions

Let's say we've got 2 users in our system: John and Sarah, where Sarah is allowed to modify a contact's last name, but John isn't. Consider the following sequence of actions:

1. John opens the "edit" view for contact "Alice Arten";
2. Sarah updates the "Alice" contact and changes her last name to "Edwards" (she got married!) and successfully saves the contact on the server;
3. John updates Alice's phone number and tries to save: the server will return an error saying he isn't authorized to modify a contact's last name. This is correct because the contact has "Edwards" as a last name (on the server), but John is still sending "Arten" to the API for her `lastName` attribute (because that's the information he has client-side after step 1). But since we're using the client-side representation to display the contact in the form, "Arten" will be the value displayed in the form accompanying the error messages saying that it can't be modified.

This situation is naturally very confusing for John, because he hasn't modified Alice's last name, even though the error message says he has. One way to avoid this confusion would be for the server to return the contact data it has at the same time that the error message is returned. Then, we could access this data to `set` the server-side values on our client-side model and avoid future errors, at the cost of losing the user's modifications in the process. To combine the two approaches and get the best of both worlds, consider the following:

1. have the API return the model representation (e.g. in an `entity` attribute) in addition to the errors when a `save` operation is unsuccessful;
2. keep a reference to `model.changedAttributes()` to know which attributes have been updated by the user;
3. reset the model's attributes to mirror the server data by calling `model.set` and passing in the server representation (i.e. `entity` in our example above);
4. reapply the changed attributes to their updated values (as saved in step 2).

Naturally if, when, and how this type of case should be handled will be up to you and the specifics of your use case.

It turns out the API we're using already returns the current server-side contact representation on error (documented earlier), so let's get started implementing this functionality…

Reapplying Server-Side Changes to the Client-Side Model

We'll begin by updating our contact to reflect the server-side data when there's an error, then reapplying the user's modifications:

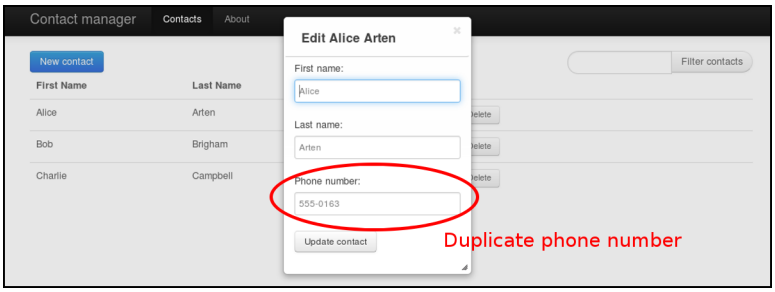
assets/js/apps/contact/list/list_controller.js

```
1 if(response.status === 422){
2   var changed = model.changedAttributes();
3   model.set(response.responseJSON.entity);
4   model.set(changed);
5   view.triggerMethod("form:data:invalid", response.responseJSON.errors);
6 }
```

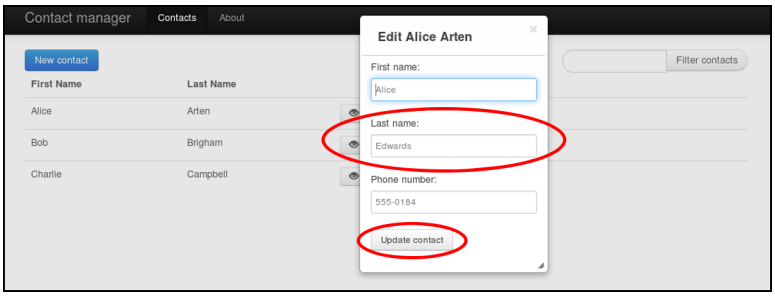
We're bascially applying the strategy we elaborated above:

1. save a reference to the changed attributes (line 2);
2. reset the model's attributes so it matches what's on the server (line 3);
3. reapply the user's modifications (line 4).

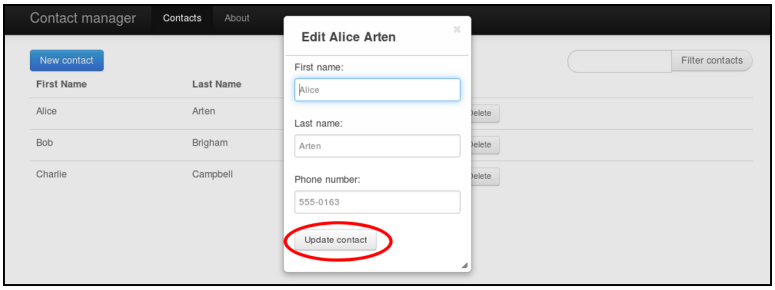
We can try it out using two tabs in our browser (both pointing to the "*#contacts*" URL with the "edit" modal open):



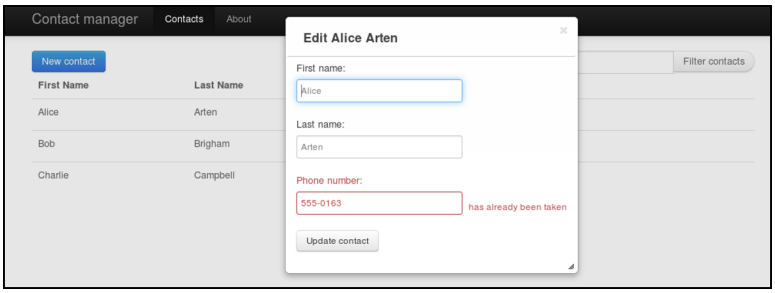
In tab 1, change a contact's phone number to a value already existing in the database (but don't save yet!)



In tab 2, change the *same* contact's last name and save it



In tab 1, attempt to save the same contact (but with a duplicate phone number)



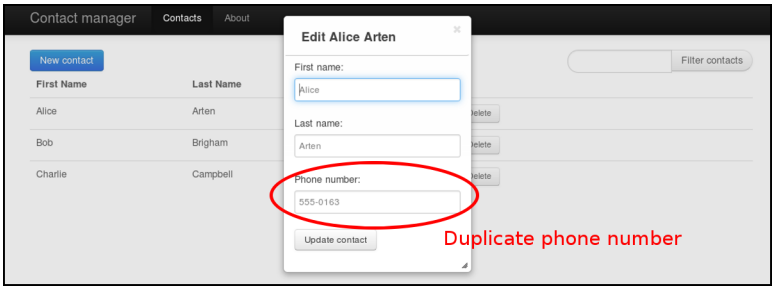
The error message is displayed, but the last name isn't updated

As you can see, the original last name is still displayed in the view even though an error message regarding the duplicate phone number is displayed. So what we need to do to get the "fresh" data from the server displayed is to force our modal view to render itself again (line 5):

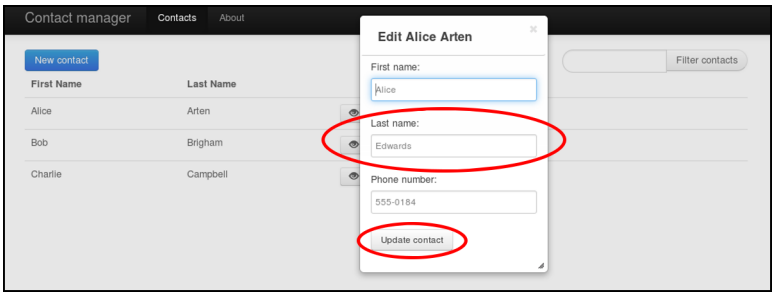
assets/js/apps/contact/list/list_controller.js

```
1  if(response.status === 422){
2    var changed = model.changedAttributes();
3    model.set(response.responseJSON.entity);
4    model.set(changed);
5    view.render();
6    view.triggerMethod("form:data:invalid", response.responseJSON.errors);
7  }
```

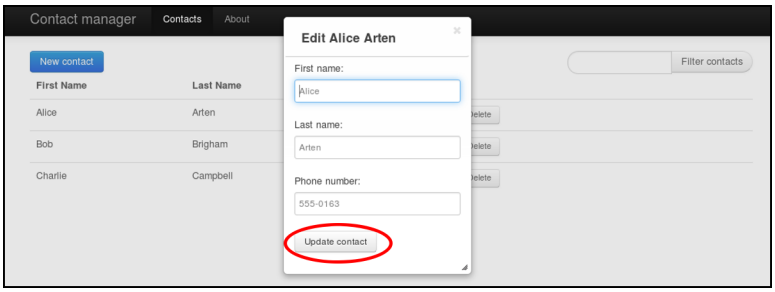
Let's try that same sequence again:



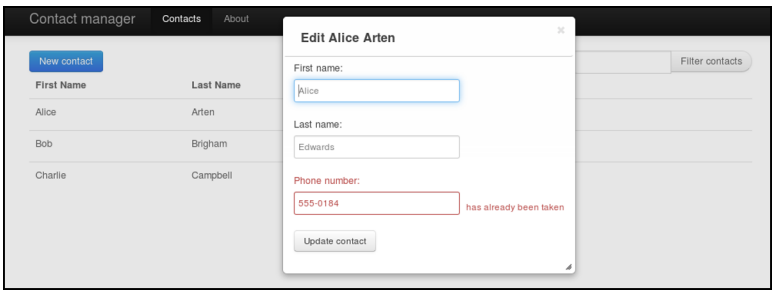
In tab 1, change a contact's phone number to a value already existing in the database (but don't save yet!)



In tab 2, change the *same* contact's last name and save it



In tab 1, save the same contact with a duplicate phone number



The error message is displayed, along with the updated last name

So we've now got something functional, but it will be confusing to the user: they're just trying to edit a contact, and values that weren't modified will change "magically". We need to add a message warning the user that the contact representation has changed on the server and has been updated locally.

Checking for Server-Side Data Changes

Let's start by setting a `changedOnServer` attribute to our model and setting it to true when there's a server-side error:

assets/js/apps/contact/list/list_controller.js

```
if(response.status === 422){
  var changed = model.changedAttributes();
  model.set(response.responseJSON.entity);
  model.set(changed);
  model.set({changedOnServer: true});
  view.render();
  view.triggerMethod("form:data:invalid", response.responseJSON.errors);
}
```

Then, based on this attribute's value, we can display a helpful message to our user in our form template:

ndex.html

```
<script type="text/template" id="contact-form">
  {{ if(changedOnServer){ }}
    <p class="alert alert-info">This model has changed on the server, and has
    been updated with the latest data from the server and your changes have
    been reapplied.</p>
  {{ } }}
  <form>
    <div class="control-group">
      <label for="contact-firstName" class="control-label">First name:</label>
      <input id="contact-firstName" name="firstName" type="text"
             value="{{- firstName }}"/>
    </div>
    <div class="control-group">
      <label for="contact-lastName" class="control-label">Last name:</label>
      <input id="contact-lastName" name="lastName" type="text"
             value="{{- lastName }}"/>
    </div>
    <div class="control-group">
      <label for="contact-phoneNumber" class="control-label">Phone number:</labe\
l>
      <input id="contact-phoneNumber" name="phoneNumber" type="text"
             value="{{- phoneNumber }}"/>
```

```
23      </div>
24      <button class="btn js-submit">Save</button>
25    </form>
26  </script>
```

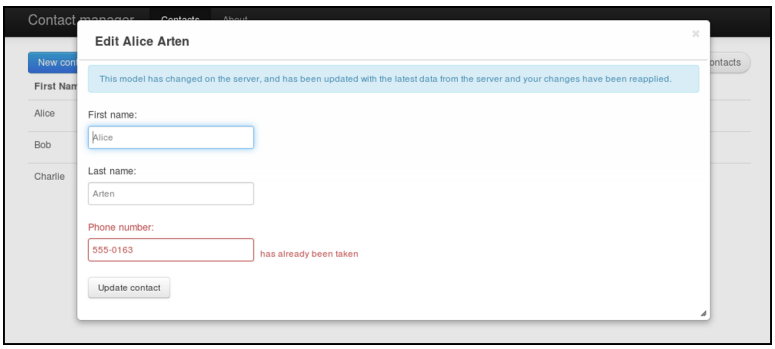
We're using the {{ and }} delimiters (as opposed to the {{- and }} we've used until now), because lines 2 and 6 don't actually output anything to the template. They're simply logic to determine whether what's between them should be processed by Underscore's templating engine.

If we attempt to try it out (by clicking on the "edit" button in the list view), we'll get an error: "ReferenceError: `changedOnServer` is not defined". That's because when we're trying to open the modal window, there's no `changedOnServer` attribute defined on the model, but the view is trying to access it. Since we'd want the value to be false unless we specifically set it to true, let's add a default value on the model:

assets/js/entities/contact.js

```
1  Entities.Contact = Entities.BaseModel.extend({
2    urlRoot: "contacts",
3
4    defaults: {
5      firstName: "",
6      lastName: "",
7      phoneNumber: "",
8
9      changedOnServer: false
10   },
11
12   // edited for brevity
```

This time around, the warning message gets properly displayed:

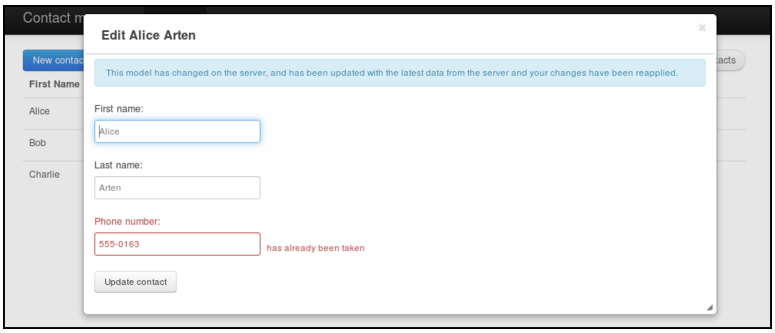


Of course, we don't actually want to display this message each time there's a server-side error: it should only be shown if the data on the server is *different* from the client-side data. Let's try doing that:

assets/js/apps/contacts/list/list_controller.js

```
if(response.status === 422){
  var serverSide = response.responseJSON.entity;
  var previousAttributes = model.previousAttributes();
  var changed = model.changedAttributes();
  model.set(serverSide);
  model.set(changed);
  model.set({
    changedOnServer: ! _.isEqual(previousAttributes, serverSide)
  });
  view.render();
  view.triggerMethod("form:data:invalid", response.responseJSON.errors);
}
```

Let's see what happens if we try updating a contact with a duplicate phone number:



So our form is notifying us that the contact data has changed on the server, although that isn't true. Why is that happeing? Let's inspect `previousAttributes` and `serverSide`:

```
// previousAttributes
{
  id: 1,
  changedOnServer: false,
  firstName: "Alice",
  lastName: "Arten",
  fullName: "Alice Arten",
  phoneNumber: "555-0184",
  url: "http://localhost:3000/contacts/1155.json"
```

```
}

// serverSide
{
  id: 1,
  firstName: "Alice",
  lastName: "Arten",
  phoneNumber: "555-0184",
  createdAt: null,
  updatedAt: null
}
```

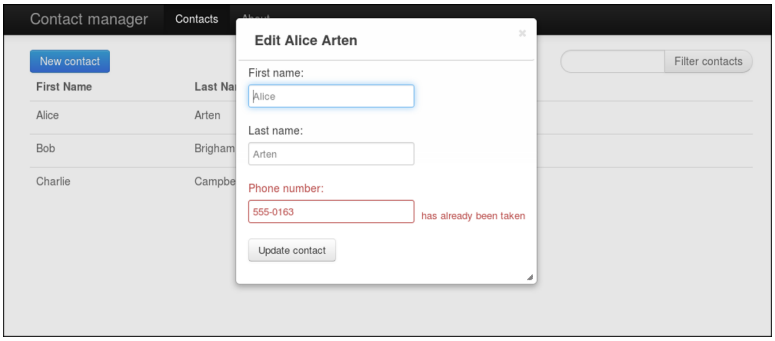
As you can see, we're comparing objects with different keys, so they're obviously considered "not equal". To address this, let's specify the attributes we want to compare:

assets/js/apps/contacts/list/list_controller.js

```
1  if(response.status === 422){
2    var serverSide = response.responseJSON.entity;
3    var previousAttributes = model.previousAttributes();
4    var changed = model.changedAttributes();
5    model.set(serverSide);
6    model.set(changed);
7    var keys = ["firstName", "lastName", "phoneNumber"];
8    var clientSide = _.pick(previousAttributes, keys);
9    var serverSide = _.pick(serverSide, keys);
10   model.set({
11     changedOnServer: ! _.isEqual(clientSide, serverSide)
12   });
13   view.render();
14   view.triggerMethod("form:data:invalid", response.responseJSON.errors);
15 }
```

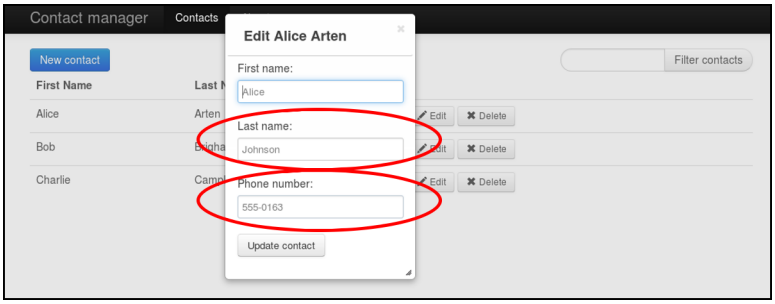
We need to store `previousAttributes` on line 3, because it will be modified when we call `set` on line 5.

With these changes in place, the message is no longer displayed when there is no change on the server:

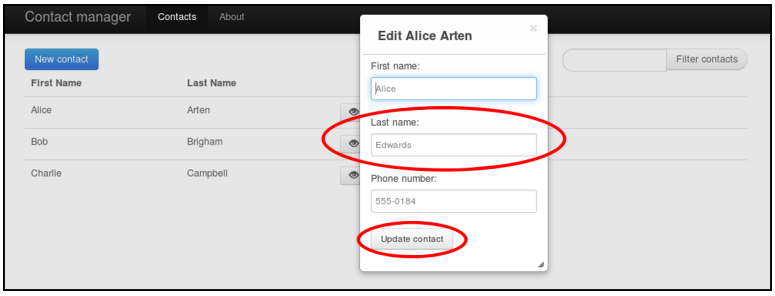


Improving the Comparison Code

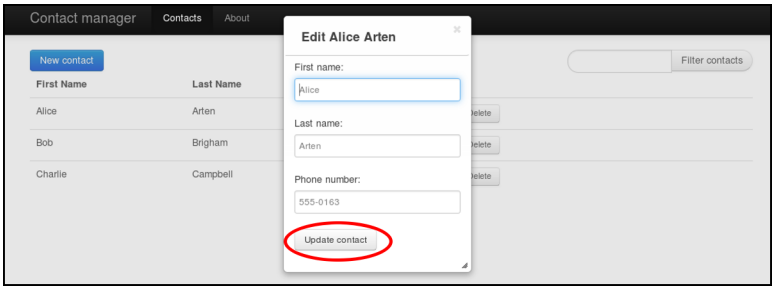
Let's improve our comparison code, so it doesn't confuse the user. Take a look at this scenario, using 2 tabs:



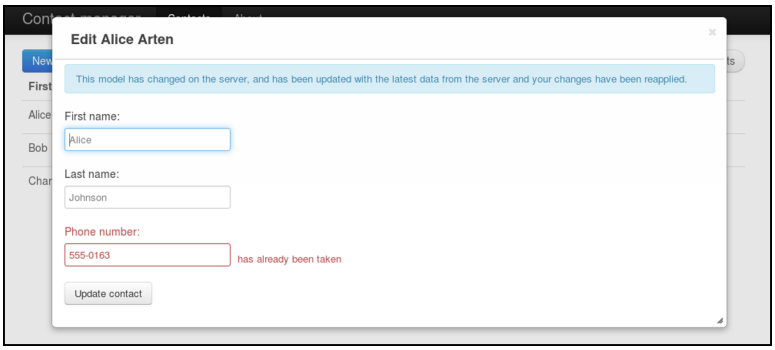
In tab 1, change a contact's last name, and the phone number to a value already existing in the database (but don't save yet!)



In tab 2, change the *same* contact's last name and save it



In tab 1, save the same contact with a duplicate phone number



The user is warned that the data has changed on the server, although it doesn't look any different

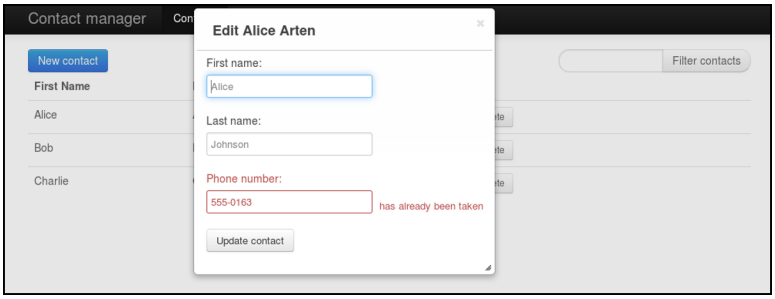
Since the user is sending "Johnson" (i.e. a new, different last name) as the value for the last name attribute, while the server has "Edwards" on disk, the warning message is displayed. However, since we're reapplying the user's client-side modifications to the data received from the server, the difference in the data is hidden to the user. To make the interface less confusing, when we're comparing the client- and server-side representations of the contact, we should ignore the attributes that the user has updated on the client:

assets/js/apps/contacts/list/list_controller.js

```
if(response.status === 422){
  var serverSide = response.responseJSON.entity;
  var previousAttributes = model.previousAttributes();
  var changed = model.changedAttributes();
  model.set(serverSide);
  var keys = ["firstName", "lastName", "phoneNumber"];
  if(changed){
    model.set(changed);
    keys = _.difference(keys, _.keys(changed));
  }
  var clientSide = _.pick(previousAttributes, keys);
  var serverSide = _.pick(serverSide, keys);
  model.set({
    changedOnServer: ! _.isEqual(clientSide, serverSide)
```

```
15    });
16    view.render();
17    view.triggerMethod("form:data:invalid", response.responseJSON.errors);
18  }
```

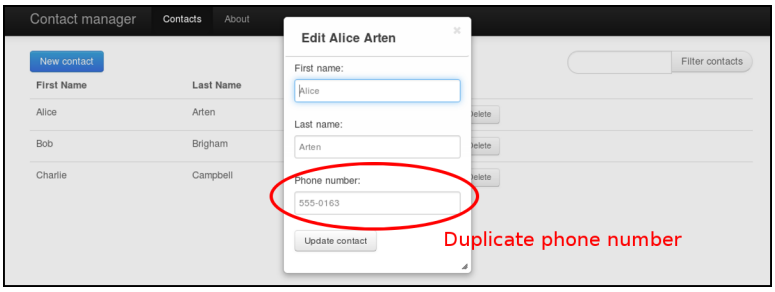
On lines 7-10, we check whether client-side attributes have changed, and remove any changed attributes from the keys we use to determine if the contact has changed on the server (line 9). The scenario above will now only display the error message next to the phone number:



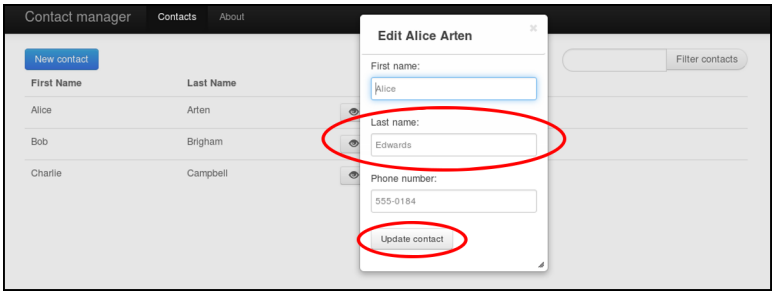
No confusing warning

Updating Client-Side Data

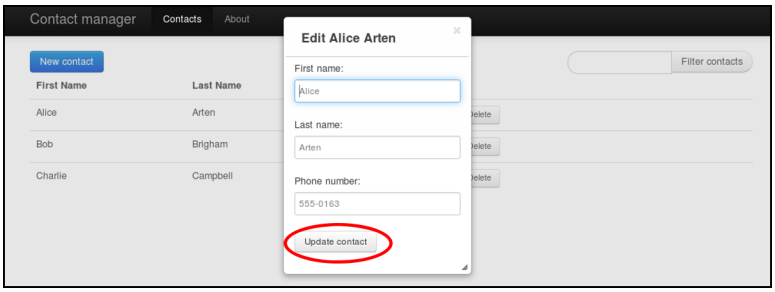
When there's a server-side error, we update the data displayed in the modal window accordingly. But the data displayed in the item view remains "stale":



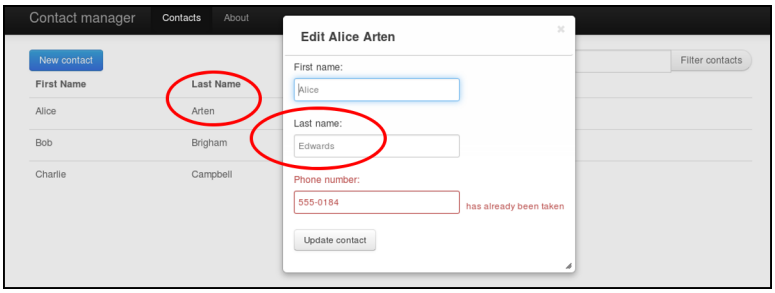
In tab 1, change a contact's phone number to a value already existing in the database (but don't save yet!)



In tab 2, change the *same* contact's last name and save it



In tab 1, save the same contact with a duplicate phone number



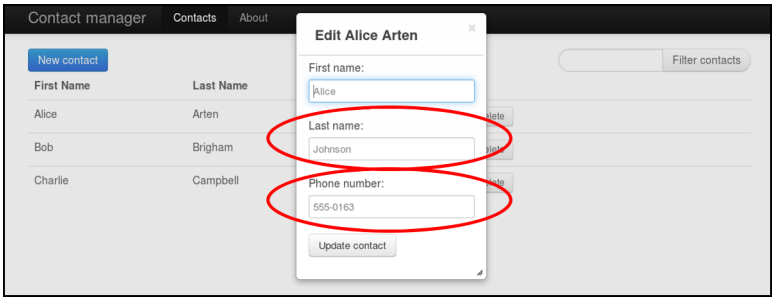
The last name is updated in the modal window, but not in the item view visible in the background

Let's have our item view render itself again anytime its model changes:

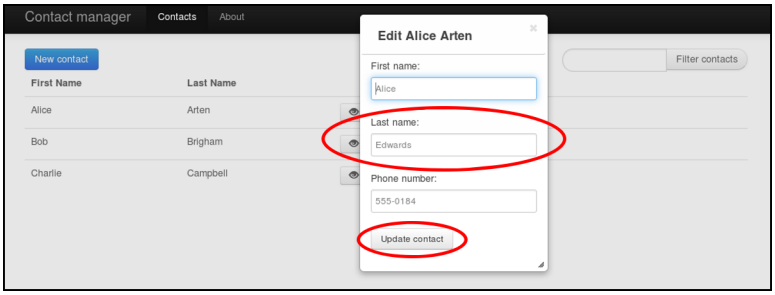
assets/js/apps/contacts/list/list_view.js

```
List.Contact = Marionette.ItemView.extend({
  tagName: "tr",
  template: "#contact-list-item",

  events: {
    // edited for brevity
  },

  modelEvents: {
    "change": "render"
  },
```

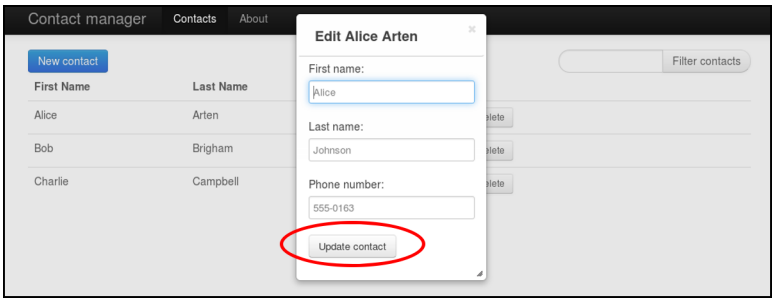
The `modelEvents` hash simply executes the function on the right when the event on the left is triggered on our model. As you can imagine, things aren't that simple: we want to update the item view with the data received from the server, but not the (unsaved) modifications from the modal view:



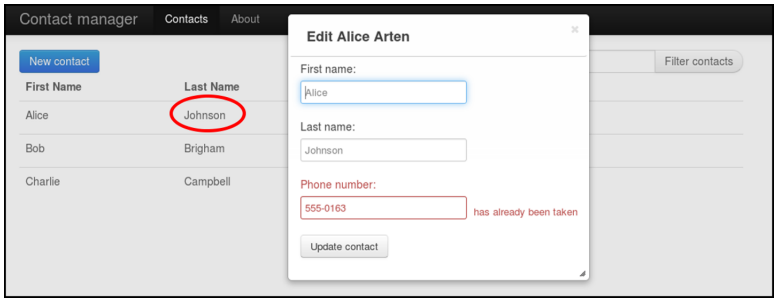
In tab 1, change a contact's phone number to a value already existing in the database, as well as the last name (but don't save yet!)



In tab 2, change the *same* contact's last name to something else and save it



In tab 1, save the same contact with a duplicate phone number



The last name is correctly updated in the modal window (with local modification reapplied), but the unsaved data has "leaked" into the item view visible in the background

That's because when we try saving the udpated last name, a "change" event is triggered, making our item view rerender itself. What we need to do is make our item view rerender itself only when we set the attributes coming from the server: any other time we call `set` it should be `silent` so it doesn't trigger a "change" event on the model (lines 8 and 15):

assets/js/app/contacts/list/list_controller.js

```
1  if(response.status === 422){
2    var serverSide = response.responseJSON.entity;
3    var previousAttributes = model.previousAttributes();
4    var changed = model.changedAttributes();
5    model.set(serverSide);
6    var keys = ["firstName", "lastName", "phoneNumber"];
7    if(changed){
8      model.set(changed, {silent: true});
9      keys = _.difference(keys, _.keys(changed));
10   }
11   var clientSide = _.pick(previousAttributes, keys);
12   var serverSide = _.pick(serverSide, keys);
13   model.set({
14     changedOnServer: ! _.isEqual(clientSide, serverSide)
15   }, {silent: true});
16   view.render();
17   view.triggerMethod("form:data:invalid", response.responseJSON.errors);
18 }
```

Refactoring

Before we address a few more issues, let's refactor the code that "refreshes" the client-side data by moving it to our base model. That way each model will inherit the functionality:

assets/js/entities/common.js

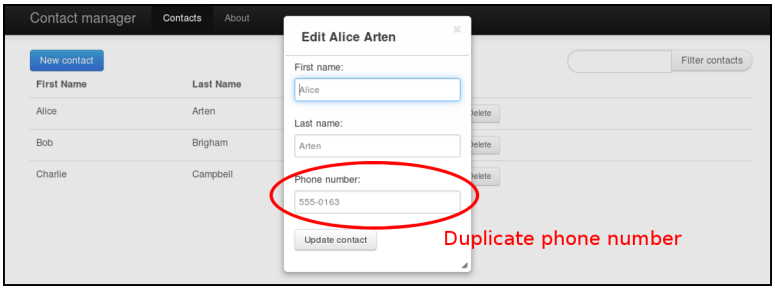
```
Entities.BaseModel = Backbone.Model.extend({
  refresh: function(serverData, keys){
    var previousAttributes = this.previousAttributes();
    var changed = this.changedAttributes();

    this.set(serverData);
    if(changed){
      this.set(changed, {silent: true});
      keys = _.difference(keys, _.keys(changed))
    }
    var clientSide = _.pick(previousAttributes, keys);
    var serverSide = _.pick(serverData, keys);
    this.set({
      changedOnServer: ! _.isEqual(clientSide, serverSide)
    }, {silent: true});
  },

  sync: function(method, model, options){
    return Backbone.Model.prototype.sync.call(this, method, model, options);
  }
});
```

assets/js/apps/contacts/list/list_controller.js

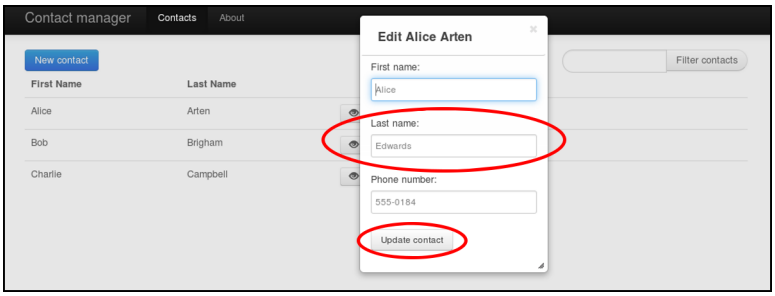
```
if(response.status === 422){
  var keys = ['firstName', 'lastName', 'phoneNumber'];
  model.refresh(response.responseJSON.entity, keys);

  view.render();
  view.triggerMethod("form:data:invalid", response.responseJSON.errors);
}
```

Properly Updating Data on Modal Dialog Close

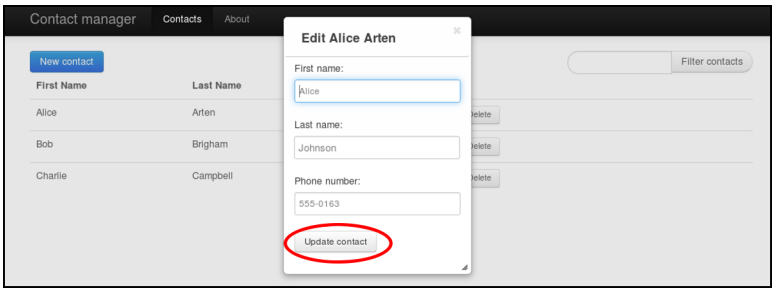
Let's investigate another bug, shall we?



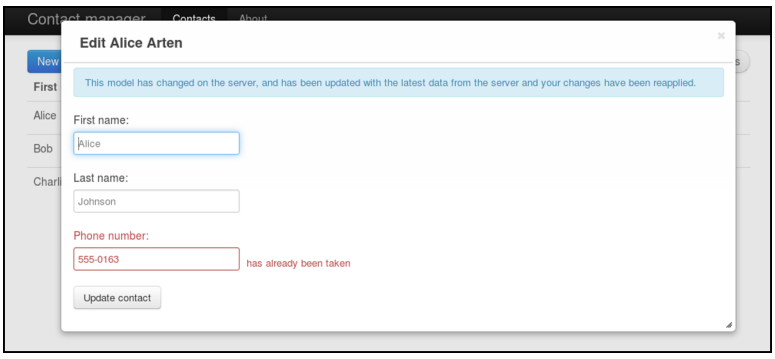
In tab 1, change a contact's phone number to a value already existing in the database (but don't save yet!)



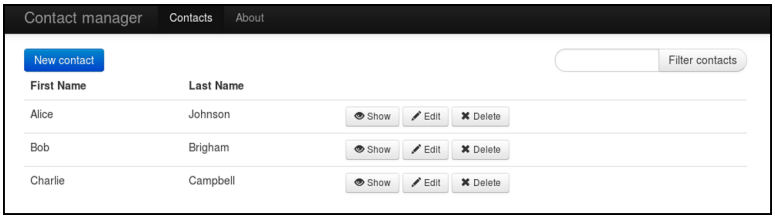
In tab 2, change the *same* contact's last name and save it (you can then close tab 2)



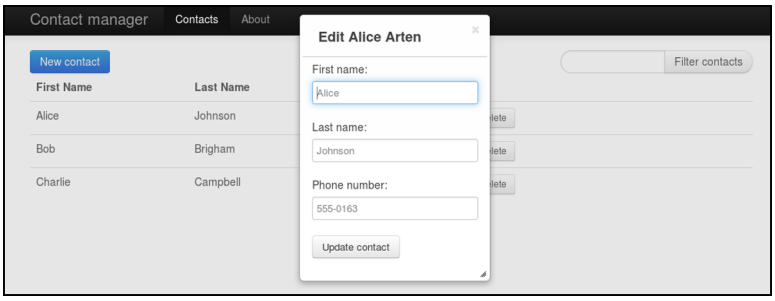
In tab 1, save the same contact with a duplicate phone number and different last name



The warning being displayed along with the changes reapplied



After closing the modal, we can see the (unsaved) changes in our modal are reflected in the item view



If we reopen the modal, we can again see that the incorrect data has been retained client-side

Let's take a look at our code to find out why it's behaving this way:

assets/js/apps/contacts/list/list_controller.js

```
1  view.on("form:submit", function(data){
2    var savingContact = model.save(data);
3    if(savingContact){
4      $.when(savingContact).done(function(){
5        // edited for brevity
6      }).fail(function(response){
7        view.onDestroy = function(){
8          model.set(model.previousAttributes());
9        };
10
11       if(response.status === 422){
12         // edited for brevity
13       }
14       else{
15         alert("An unprocessed error happened. Please try again!");
16       }
17     });
18   }
19   else{
20     view.triggerMethod("form:data:invalid", model.validationError);
21   }
22 });
```

On lines 7-9 we set the model's attributes to their previous values when the modal window is closed. Unfortunately, in the case where we refresh the data with the information coming from the server, that previous state doesn't match what is saved on the server and includes unsaved client-side updates. Let's fix that by defining 2 different `onDestroy` methods:

assets/js/apps/contacts/list/list_controller.js

```
1  view.on("form:submit", function(data){
2    var savingContact = model.save(data);
3    if(savingContact){
4      $.when(savingContact).done(function(){
5        childView.render();
6        delete view.onDestroy;
7        view.trigger("dialog:close");
8        childView.flash("success");
9      }).fail(function(response){
10       if(response.status === 422){
11         view.onDestroy = function(){
12           model.set(response.responseJSON.entity);
13         };
14
15         // edited for brevity
16       }
17       else{
18         alert("An unprocessed error happened. Please try again!");
19       }
20     });
21   }
22   else{
23     view.onDestroy = function(){
24       model.set(model.previousAttributes());
25     };
26
27     view.triggerMethod("form:data:invalid", model.validationError);
28   }
29 });
```

We define our appropriate `onDestroy` methods on lines 11-13 and 23-25, which we need to "undo" on line 6. Without that line, the following scenario wouldn't work properly:

1. open the modal "edit" dialog and change the phone number to a duplicate value;
2. try saving the data: you'll be unable to due to the duplicate phone number;

3. delete the duplicate phone number in the form and update the contact's last name;
4. save the data (which will be successful, since there are no errors);
5. you'll see that the item view reverts to displaying the old data (i.e. your updated data isn't displayed).

This is because on step 2 our code will define an `onDestroy` method to set our model data to what it received from the server. Then, when our contact is successfully saved in step 4, the `onDestroy` method will be executed when we close the modal dialog by triggering the "dialog:close" event.

A related issue arises in the following use case:

1. open the modal "edit" dialog and change the phone number to a duplicate value;
2. in another tab, change the contact's last name and save it (you can then close tab 2);
3. back in tab 1, try saving the data: you'll be unable to due to the duplicate phone number;
4. delete the duplicate phone number and save the contact;
5. open the "edit" modal dialog again and you will see the "data changed on server" warning, which shouldn't be displayed.

To fix this, we simply need to reset the `changedOnServer` attribute before the modal is closed:

assets/js/apps/contacts/list/list_controller.js

```
1       view.on("form:submit", function(data){
2         var savingContact = model.save(data);
3         if(savingContact){
4           view.onBeforeClose = function(){
5             model.set({changedOnServer: false});
6           };
7
8           // edited for brevity
```

Fixing a UI Glitch

We're nearing a bug-free implementation of our functionality, but we still have UI glitch where incorrect data is being displayed in the UI before being "fixed":

1. open the "edit" modal, change the contact's last name, and enter a duplicate phone number;
2. make sure you move the modal window so that you can see the related item view in the background;
3. click save while looking at the item view's last name;
4. you'll see that the unsaved last name gets displayed briefly in the item view before being reverted to the (previously) saved value.

Look at our current code:

assets/js/apps/contacts/list/list_controller.js

```
1  view.on("form:submit", function(data){
2    var savingContact = model.save(data);
3    if(savingContact){
4      // edited for brevity
5      $.when(savingContact).done(function(){
6        // edited for brevity
7      }).fail(function(response){
8        if(response.status === 422){
9          // edited for brevity
10
11         var keys = ['firstName', 'lastName', 'phoneNumber'];
12         model.refresh(response.responseJSON.entity, keys);
13
14         view.render();
15         view.triggerMethod("form:data:invalid", response.responseJSON.errors);
16       }
17       else{
18         alert("An unprocessed error happened. Please try again!");
19       }
20     });
21   }
22   else{
23     // edited for brevity
24   }
25 });
```

Here's what happens when we try to update a contact with incorrect data:

1. on line 2, we attempt to save our updated model, which triggers a "set" event;
2. our item view reacts to the "set" event and rerenders itself with the unsaved data;
3. in our `refresh` call on line 12 we restore the correct values by triggering `set` calls, and silencing the ones we don't want to trigger an item view rerendering.

With that in mind, we need to fix our problem by making Backbone wait for the server response before setting the new attributes on the model:

assets/js/apps/contacts/list/list_controller.js

```
1  view.on("form:submit", function(data){
2    model.set(data, {silent: true});
3    var savingContact = model.save(data, {wait: true});
4    if(savingContact){
5      // edited for brevity
6      $.when(savingContact).done(function(){
7        // edited for brevity
8      }).fail(function(response){
9        if(response.status === 422){
10         // edited for brevity
11
12         var keys = ['firstName', 'lastName', 'phoneNumber'];
13         model.refresh(response.responseJSON.entity, keys);
14
15         view.render();
16         view.triggerMethod("form:data:invalid", response.responseJSON.errors);
17         model.set(response.responseJSON.entity, {silent:true});
18       }
19       else{
20         alert("An unprocessed error happened. Please try again!");
21       }
22     });
23   }
24   else{
25     // edited for brevity
26   }
27 });
```

On line 2, we `set` the new attributes silently, which will behave exactly as we want it to: it will set the new attribute values on the model (so they'll be displayed in the view), but the item view won't be rerendered because no "set" event will be triggered. Then, we change our `save` call on line 3 to `wait` for the server response before setting the updated attribute values on the model. And there: the UI glitch is gone!

One last thing we need to do is to once again silently `set` the model attributes on line 17 so that our next call to (e.g.) `changedAttributes` will provide us with the result we're looking for: the changes that have been made to the model client side (which are determined based on the last `set` call).

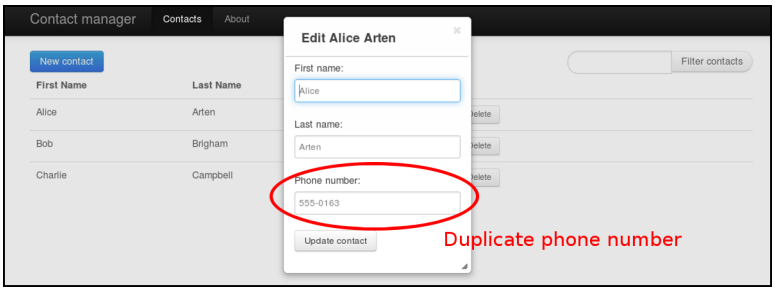
Git commit refreshing client data on error

806038fad879c68653d0965099ca81d6f6e6e0db[81]

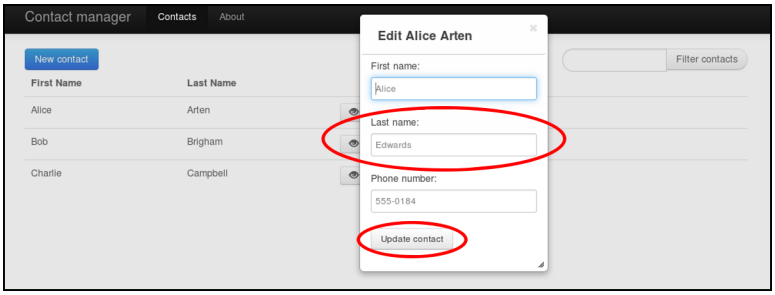
[81] https://github.com/davidsulc/marionette-serious-progression-806038fad879c68653d0965099ca81d6f6e6e0db

Keeping `fullName` Up to Date

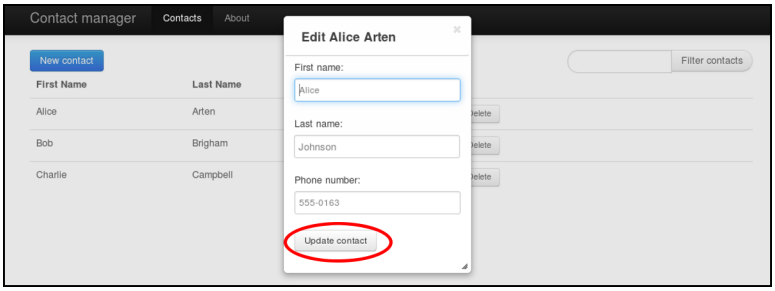
We've still got a small annoyance in our UI: when we receive up-to-date server-side information, our model properly updates the `firstName`, `lastName`, and `phoneNumber` attributes (among others). Our computed `fullName` attribute remains unchanged, and the incorrect previous name is displayed as the modal's title:



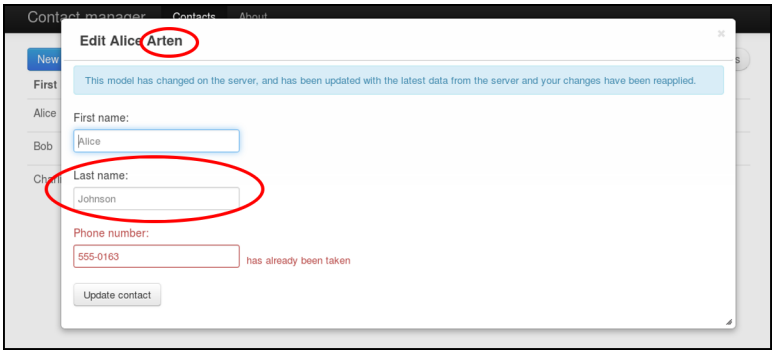
In tab 1, change a contact's phone number to a value already existing in the database (but don't save yet!)



In tab 2, change the *same* contact's last name and save it (you can then close tab 2)



In tab 1, save the same contact with a duplicate phone number and different last name



The old displayed in the title, while updated in the text field

We will see how to keep our computed attribute updated, but be aware that this example isn't the best example of when you should use virtual attributes. In particular, be aware there is an associated cost and only use them if that cost is justified. Simply displaying a full name in a title, for example, would be better implemented as

```
1  <h1>{{- firstName }} {{- lastName }}</h1>
```

To keep our `fullName` attribute up to date, we simply need to recompute it when the model changes:

assets/js/entities/contact.js

```
1  Entities.Contact = Entities.BaseModel.extend({
2    urlRoot: "contacts",
3
4    initialize: function(){
5      this.on("change", function(){
6        this.set("fullName", this.get("firstName") + " " + this.get("lastName"));
7      });
8    },
9
10   // edited for brevity
```

Let's take a quick look at our dialog region manager to see how the title is retrieved for display:

assets/js/apps/config/marionette/region/dialog.js

```
1  Marionette.Region.Dialog = Marionette.Region.extend({
2    onShow: function(view){
3      this.listenTo(view, "dialog:close", this.closeDialog);
4
5      var self = this;
6      this.$el.dialog({
7        modal: true,
8        title: view.title,
9        width: "auto",
10       close: function(e, ui){
11         self.closeDialog();
12       }
13     });
14   },
15
16   closeDialog: function(){
17     this.stopListening();
18     this.close();
19     this.$el.dialog("destroy");
20   }
21 });
```

On line 8, we get the view's `title` attribute to use as the modal's title. Therefore, we need to make sure that view attribute is updated before we try to render the view (instead of just once when the view is initialized):

assets/js/apps/contacts/edit/edit_view.js

```
1  Edit.Contact = ContactManager.ContactsApp.Common.Views.Form.extend({
2  ~~  initialize: function(){~~
3    onBeforeRender: function(){
4      this.title = "Edit " + this.model.get("fullName");
5    },
6
7    // edited for brevity
```

If you try our scenario again, it still won't work: our dialog region configures the dialog the first time the view is rendered (which sets the title), and then simply rerenders the contents without affecting the title. This is also why the dialog will no longer be centered when an error message is displayed and the dialog width changes. Let's address that in the next section.

Fixing the Modal Display

What we need to do is configure our dialog (set the title, position, etc.) before it gets displayed, but also reconfigure it each time the contained view is rerendered:

assets/js/apps/config/marionette/region/dialog.js

```
1  Marionette.Region.Dialog = Marionette.Region.extend({
2    onShow: function(view){
3      this.listenTo(view, "dialog:close", this.closeDialog);
4
5      var self = this;
6      var configureDialog = function(){
7        self.$el.dialog({
8          modal: true,
9          title: view.title,
10         width: "auto",
11         position: "center",
12         close: function(e, ui){
13           self.closeDialog();
14         }
15       });
16     };
17     configureDialog();
18
19     this.listenTo(view, "render", configureDialog);
20   },
21
22   closeDialog: function(){
23     // edited for brevity
24   }
25 });
```

On lines 6-16, we define a `configureDialog` function which simply reconfigures the dialog as we want it. We then call this function on line 17 to configure the dialog when a view is shown within the region. In addition, we also need to call it each time the contained view is rendered (line 19).

But before our code is fully functional, we need to make a small change in our view's `onRender` function so it will behave properly if no options are provided (line 2):

assets/js/apps/contacts/edit/edit_view.js

```
onRender: function(){
  if(this.options && this.options.generateTitle){
    var $title = $('<h1>', { text: this.title });
    this.$el.prepend($title);
  }

  this.$(".js-submit").text("Update contact");
}
```

Git commit fixing the UI issues

07272473f583196d45bdfa843a4202d33e295fe2[82]

UI Challenges Aren't so Simple

In this chapter we dealt with implementing a relatively simple functionality: update the client-side data based on information returned by the server. As we saw, a simple description like this actually entails a sizeable amount of work and details that have to be dealt with. To end up with an enjoyable user interface (i.e. that doesn't surprise or confuse the user), it's important to understand how data management on the client affects the user interface, and to link the individual pieces together properly. Mainly, you need to pay very close attention to (e.g.) `set` calls and when the views should be updated (i.e. if `silent` should be `true` or not). In addition, you must also be clear on the state your model is in at any given time (e.g. what `previousAttributes` will return).

The same type of logic and code could be applied (e.g.) to notify a user that server-side data has changed for that model *before* accepting an update (even if it contains no errors). But again: this type of feature requires attention to detail, and making sure that the changes made to the client-side data are controlled to only trigger view updates when appropriate.

Try your hand at this type of thinking by implementing the client-side data "refreshing" to *assets/js/apps/contacts/edit/edit_controller.js*. Although it will be useful to refer to our above implementation, you won't be able to simply copy and paste: the UI is different (there's no modal window, and no exposed item views), so it requires different data handling. Continue to the next page to compare your solution.

[82] https://github.com/davidsulc/marionette-serious-progression-07272473f583196d45bdfa843a4202d33e295fe2

assets/js/apps/contacts/edit/edit_controller.js

```
1  view.on("form:submit", function(data){
2    contact.set(data, {silent: true});
3    var savingContact = contact.save(data);
4    var savingContact = contact.save(data, {wait: true});
5    if(savingContact){
6      view.onBeforeClose = function(){
7        contact.set({changedOnServer: false});
8      };
9      $.when(savingContact).done(function(){
10       ContactManager.trigger("contact:show", contact.get("id"));
11     }).fail(function(response){
12       if(response.status === 422){
13         var keys = ['firstName', 'lastName', 'phoneNumber'];
14         contact.refresh(response.responseJSON.entity, keys);
15
16         view.render();
17         view.triggerMethod("form:data:invalid", response.responseJSON.errors);
18         contact.set(response.responseJSON.entity, {silent:true});
19       }
20       else{
21         alert("An unprocessed error happened. Please try again!");
22       }
23     });
24   }
25   else{
26     view.triggerMethod("form:data:invalid", contact.validationError);
27   }
28 });
```

Note that on lines 6-8 we still need to reset the `changedOnServer` attribute so the warning doesn't get displayed when it shouldn't be. In addition, we don't have any code related to closing the modal window and making sure the proper data is displayed in the item views listing all the contacts.

Git commit fixing the normal edit controller to handle server data refresh 4843875bcb96cab1e3615b91552af25c557b4174[83]

[83] https://github.com/davidsulc/marionette-serious-progression-4843875bcb96cab1e3615b91552af25c557b4174

Handling Validation with a Plugin

So far we've been handling validation manually by defining a `validate` method on our model, and displaying validation errors on our form. In this chapter, we'll see how we can offload this work to a plugin instead, by using Backbone.Validation[84].

Let's get the ball rolling by saving the plugin file from here[85] to *assets/js/vendor/backbone.validation.js*, and requiring it in our index file:

index.html

```
<script src="./assets/js/vendor/jquery.js"></script>
<script src="./assets/js/vendor/jquery-ui-1.10.3.js"></script>
<script src="./assets/js/vendor/json2.js"></script>
<script src="./assets/js/vendor/underscore.js"></script>
<script src="./assets/js/vendor/backbone.js"></script>
<!-- new addition -->
<script src="./assets/js/vendor/backbone.validation.js"></script>
<script src="./assets/js/vendor/backbone.picky.js"></script>
<script src="./assets/js/vendor/backbone.syphon.js"></script>
<script src="./assets/js/vendor/backbone.marionette.js"></script>
<script src="./assets/js/vendor/spin.js"></script>
<script src="./assets/js/vendor/spin.jquery.js"></script>
```

Don't forget that since Backbone.Validation is a Backbone plugin, it must be included *after* the Bacbkone file.

Model-Only Validation

Let's start by using the plugin only for model validation: we'll continue to handle the view update (i.e. adding error messages) by hand. We need to replace our `validate` method with a `validation` configuration object:

[84] https://github.com/thedersen/backbone.validation

[85] https://github.com/davidsulc/marionette-serious-progression-app/public/assets/js/vendor/backbone.validation.js

assets/js/entities/contact.js

```
// edited for brevity

validate: function(attrs, options) {
  var errors = {}
  if (! attrs.firstName) {
    errors.firstName = "can't be blank";
  }
  if (! attrs.lastName) {
    errors.lastName = "can't be blank";
  }
  else{
    if (attrs.lastName.length < 2) {
      errors.lastName = "is too short";
    }
  }
  if( ! _.isEmpty(errors)){
    return errors;
  }
},

validation: {
  firstName: {
    required: true
  },

  lastName: {
    required: true,
    minLength: 2
  }
},

// edited for brevity
```

The structure and content of the `validation` object are naturally dictated by the plugin (see documentation[86]). Of course, the plugin offers many other built-in validators[87] to chose from.

Now that we've defined how our model should be validated, we still need to indicate that our `Contact` model "class" should add the validation plugin functionality to itself. To do so, we need to extend the `Contact` prototype:

[86] https://github.com/thedersen/backbone.validation#configure-validation-rules-on-the-model

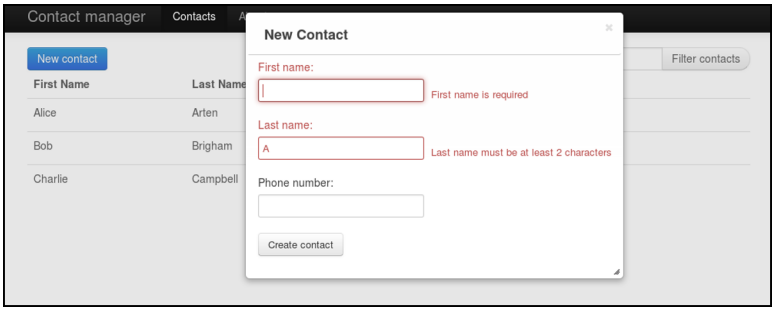
[87] https://github.com/thedersen/backbone.validation#built-in-validators

assets/js/entities/contact.js

```
Entities.Contact = Entities.BaseModel.extend({
  // edited for brevity
});

_.extend(Entities.Contact.prototype, Backbone.Validation.mixin);
```

We've mentioned javascript prototypes here and there, but they will be covered in more depth in chapter Memory Management.

Our model is now being validated using the validation plugin!



The form displaying Backbone.Validation's default error messages

Defining Custom Error Messages

To define custom error messages, we simply need to extend the plugin's `messages` object:

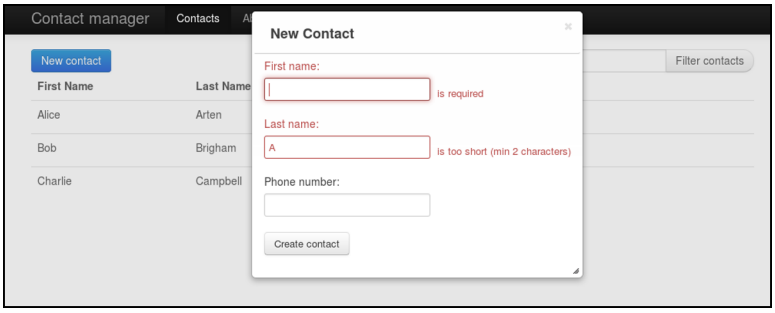
assets/js/entities/common.js

```
_.extend(Backbone.Validation.messages, {
  required: 'is required',
  minLength: 'is too short (min {1} characters)'
});
```

We've only modified the error messages we'll actually be using, but you can override the other messages too (refer to the plugin's documentation[88]). Another option is to specify the error message you want to when you're defining the validation (see here[89]).

[88] https://github.com/thedersen/backbone.validation#extending-backbone-validation
[89] https://github.com/thedersen/backbone.validation#specifying-error-messages

Take a look at our custom error messages in action:



The form with our custom error messages

Git commit using validation plugin to validate contact model (with custom error messages) b0e4547aaecd57d0e5535270142b7bbf4d7ec6d4[90]

Form + Model Validation

The view binding explained in this chapter won't be included in our application, because it's difficult to integrate with non-client side error generation (e.g. when receiving errors from the server). In other words, we will continue updating the form errors using our current method, because it allows us to transparently display client- and server-side validation errors.

You can have your form display the model's error messages using Backbone.Validator's functionality. To do so, you'll need to bind to the view before it is rendered, and unbind from it once it is no longer displayed:

assets/js/apps/contacts/common/views.js

```
1  Views.Form = Marionette.ItemView.extend({
2    template: "#contact-form",
3
4    initialize: function(){
5      Backbone.Validation.bind(this);
6    },
7
8    onBeforeClose: function(){
9      Backbone.Validation.unbind(this);
```

[90] https://github.com/davidsulc/marionette-serious-progression-b0e4547aaecd57d0e5535270142b7bbf4d7ec6d4

```
  },

  // edited for brevity
```

With this code in place, when the plugin tries to validate the model and it is invalid, the plugin will (per the documentation[91])

> try to look up an element within the view with a name attribute equal to the name of the attribute that is validated. If it finds one, an invalid class is added to the element as well as a data-error attribute with the error message

Since it is recommended to provide your own implementation of how to handle the invalid and valid models, you can specify them when binding:

assets/js/apps/contacts/common/views.js

```
Views.Form = Marionette.ItemView.extend({
  template: "#contact-form",

  initialize: function(){
    Backbone.Validation.bind(this, {
      valid: function(view, attr){
        // remove errors on the `attr` attribute from the view
      },
      invalid: function(view, attr, error){
        // add error on the `attr` attribute to the view
      }
    });
  },

  onBeforeClose: function(){
    Backbone.Validation.unbind(this);
  },

  // edited for brevity
```

Note that `invalid` gets called for each error in sequence. Also, the `valid` method will be called between 2 "invalid model" events to remove all errors from the view (they will be added again right after, with a call to `invalid`).

[91] https://github.com/thedersen/backbone.validation#callbacks

Pagination

We're going to need lots of contacts to demonstrate pagination, so let's have some created. If you've got Rails installed locally, go into your project's top-level directory (in a terminal) and execute:

```
rake fake:contacts
```

If you're working remotely on Heroku, execute the following from the terminal (see documentation[92]):

```
heroku run rake fake:contacts
```

If you're following along with your own API, simply generate 503 fake contacts in your database. The odd number is so the you can verify that the last page behaves correctly by displaying a shorter number of contacts.

In this chapter, we'll take a look a pagination, using Addy Osmani's Backbone.Paginator plugin[93]. We'll tackle 3 main challenges:

- build a `PaginatedView` that will turn any provided collection and composite view into a paginated view, along with page controls;
- use client-side pagination, which is purely a user-interface implementation: the entire collection gets fetched from the server, but only a portion is displayed at any given time;
- use server-side pagination: each time a user changes the results page (e.g. by clicking on the page controls), only the models for that partcular page are fetched from the server.

But first, grab the library from here[94], put it in *assets/js/vendor/backbone.paginator.js*, and add it to the index page includes:

[92] https://devcenter.heroku.com/articles/rake

[93] https://github.com/backbone-paginator/backbone.paginator

[94] https://github.com/davidsulc/marionette-serious-progression-app/public/assets/js/vendor/backbone.paginator.js

index.html

```
<script src="./assets/js/vendor/jquery.js"></script>
<script src="./assets/js/vendor/jquery-ui-1.10.3.js"></script>
<script src="./assets/js/vendor/json2.js"></script>
<script src="./assets/js/vendor/underscore.js"></script>
<script src="./assets/js/vendor/backbone.js"></script>
<script src="./assets/js/vendor/backbone.validation.js"></script>
<!-- add the following line -->
<script src="./assets/js/vendor/backbone.paginator.js"></script>
<script src="./assets/js/vendor/backbone.picky.js"></script>
<script src="./assets/js/vendor/backbone.syphon.js"></script>
<script src="./assets/js/vendor/backbone.marionette.js"></script>
<script src="./assets/js/vendor/spin.js"></script>
<script src="./assets/js/vendor/spin.jquery.js"></script>
```

Using ClientPager

We begin by using a paginated collection in our entity file:

assets/js/entities/contact.js

```
~~Entities.ContactCollection = Backbone.Collection.extend({~~
Entities.ContactCollection = Backbone.Paginator.clientPager.extend({
  ~~url: "contacts",~~
  model: Entities.Contact,
  comparator: "firstName",
  paginator_core: {
    dataType: "json",
    url: "contacts"
  },
  paginator_ui: {
    firstPage: 1,
    currentPage: 1,
    perPage: 10,
    pagesInRange: 2
  }
 });
```

We're converting the collection we use in the app to a paginated collection, because everywhere we'll use a list of contacts, we'll want it to be paginated. In addition, as we later implement the `requestPager`, it will enable us to query only the first page of results instead of *all* the contacts in the database. If we only wanted some collection instances to be paginated, we could instead instantiate the paginated collections locally, e.g.:

```
1  var PaginatedCollection = Backbone.Paginator.clientPager.extend({
2    // edited for brevity
3  });
4
5  var paginatedCollection = new PaginatedCollection(
6                                      nonPaginatedCollection.models
7                                    );
```

We also need to make a small change in our request handler code, so that the paginated collection works properly (line 7):

assets/js/entities/contact.js

```
1  getContactEntities: function(options){
2    var contacts = new Entities.ContactCollection();
3    var defer = $.Deferred();
4    options || (options = {});
5    // for paginator,
6    // see https://github.com/backbone-paginator/backbone.paginator/pull/180
7    options.reset = true;
8    defer.then(options.success, options.error);
9
10   // edited for brevity
```

We can now adapt our controller code to use this new collection: since paginator also provides filtering functionality, we'll no longer use our `FilteredCollection`:

assets/js/apps/contacts/list/list_controller.js

```
$.when(fetchingContacts).done(function(contacts){
var filteredContacts = ContactManager.Entities.FilteredCollection({
  collection: contacts,
  filterFunction: function(filterCriterion){
    var criterion = filterCriterion.toLowerCase();
    return function(contact){
      if(contact.get("firstName").toLowerCase().indexOf(criterion) !== -1
        || contact.get("lastName").toLowerCase().indexOf(criterion) !== -1
        || contact.get("phoneNumber").toLowerCase().indexOf(criterion) !== -1){
          return contact;
      }
    };
  }
});

if(criterion){
  filteredContacts.filter(criterion);
  contacts.filter(criterion);
  contactsListPanel.once("show", function(){
    contactsListPanel.triggerMethod("set:filter:criterion", criterion);
  });
}

var contactsListView = new List.Contacts({
  collection: filteredContacts
  collection: contacts
});

contactsListPanel.on("contacts:filter", function(filterCriterion){
  filteredContacts.filter(filterCriterion);
  contacts.filter(filterCriterion);
  ContactManager.trigger("contacts:filter", filterCriterion);
});
```

Note that, at this time, filtering functionality isn't impelemented! If you try filtering either through the interface or by using a URL parameter, the application will raise an error. This functionality will be reimplemented later, and this code is left as is to make things easier then.

Our app will now fetch the collection from the server, but as you can see, the entire contents of the collection is displayed. Where's our pagination? Since we're using the client pager, all of the contacts

get loaded on `fetch`. So what we need to do, once we have the contacts, is to make the paginated collection go to the first page of result before we display it (line 1):

assets/js/apps/contacts/list/list_controller.js

```
1  contacts.goTo(1);
2  var contactsListView = new List.Contacts({
3    collection: contacts
4  });
```

Building a Generic Paginated View

Let's add some pagination controls to our paginated view, so that users can see how many pages of content are available, and navigate through them.

First, let's pretend we already have a view to manage our pagination controls (we'll once again define it a little farther down). We'll need to instantiate and display the pagination view when our main layout gets initialized:

assets/js/common/views.js

```
1  Views.PaginatedView = Marionette.LayoutView.extend({
2    template: "#paginated-view",
3
4    regions: {
5      paginationControlsRegion: ".js-pagination-controls",
6      paginationMainRegion: ".js-pagination-main"
7    },
8
9    initialize: function(options){
10     this.collection = options.collection;
11
12     var controls = new Views.PaginationControls({
13       paginatedCollection: this.collection
14     });
15     var listView = new options.mainView({
16       collection: this.collection
17     });
18
19     this.on("show", function(){
20       this.paginationControlsRegion.show(controls);
21       this.paginationMainRegion.show(listView);
22     });
```

```
23    }
24  });
```

So let's define our pagination controls view:

assets/js/common/views.js

```
1  Views.PaginationControls = Marionette.ItemView.extend({
2    template: "#pagination-controls",
3    className: "pagination",
4
5    initialize: function(options){
6      this.paginatedCollection = options.paginatedCollection;
7    },
8
9    serializeData: function(){
10     return _.clone(this.paginatedCollection.info());
11   }
12 });
```

As you can see on line 5, this sub-view will expect a paginated collection to be provided to it within the `paginatedCollection` attribute. In order to display the pagination controls properly (current page, next page, etc.) we'll need information on the current state of pagination, which we take from the `info` method provided by Backbone.Paginator, and we send that information on to the view (lines 9-11). We make our element use the `pagination` class name (line 3), so that Bootstrap can style the pagination controls for us (documentation[95]).

The `info` method is internal to Backbone.Paginator, and therefore it isn't exactly best practice to access it directly. However, it is much more convenient than having to define the current page, next page, and other attributes manually before passing them on to the view. Just be aware that Backbone.Paginator could break this by changing its internal implementation…

Last, but definitely not least, we need to have a template to display:

[95] http://getbootstrap.com/2.3.2/components.html#pagination

index.html

```
<script type="text/template" id="pagination-controls">
  {{ if(totalPages > 1){ }}
    <ul>
      {{ if(currentPage > 1){ }}
        <li><a href="#">&laquo;</a></li>
        <li><a href="#">&lsaquo;</a></li>
      {{ }else{ }}
        <li class="disabled"><a href="#">&laquo;</a></li>
        <li class="disabled"><a href="#">&lsaquo;</a></li>
      {{ } }}

      {{ if(pageSet[0] > 1 ){ }}
        <li class="disabled"><a href="#">...</a></li>
      {{ } }}

      {{ _.each(pageSet, function(page){ }}
        {{ if(page === currentPage){ }}
          <li class="active"><a href="#">{{- page }}</a></li>
        {{ }else{ }}
          <li><a href="#">{{- page }}</a></li>
        {{ } }}
      {{ }); }}

      {{ if(pageSet[pageSet.length - 1] !== lastPage){ }}
        <li class="disabled"><a href="#">...</a></li>
        <li><a href="#">{{- lastPage }}</a></li>
      {{ } }}

      {{ if(currentPage !== lastPage){ }}
        <li><a href="#">&rsaquo;</a></li>
        <li><a href="#">&raquo;</a></li>
      {{ }else{ }}
        <li class="disabled"><a href="#">&rsaquo;</a></li>
        <li class="disabled"><a href="#">&raquo;</a></li>
      {{ } }}
    </ul>
  {{ } }}
</script>
```

The main characteristics of the template are:

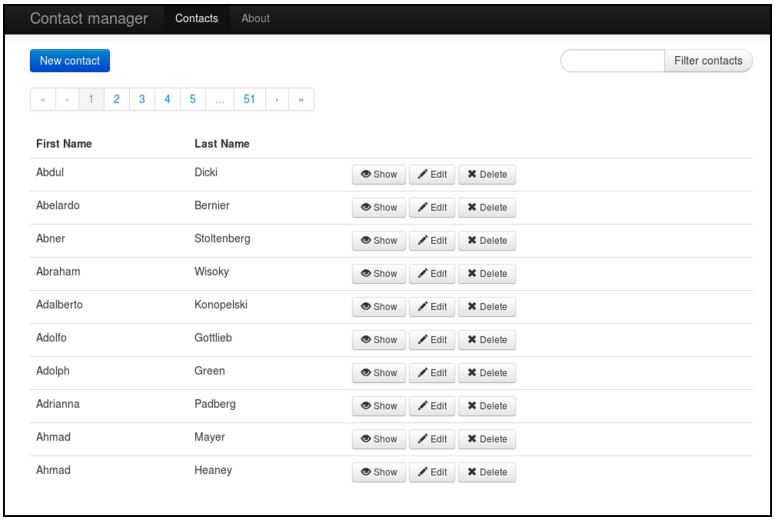
- if there's only one page, don't display the controls (line 1);
- only activate the "first page" / "previous page" and "next page" / "last page" links if we aren't one the first or last page, respectively (lines 4-10 and 29-35);
- if there's a section of pages for which we're not displaying direct links, add "..." to indicate so (lines 12-14 and 24-27);
- finally, iterate through the accessible page range we've defined on the collection (via the `pagesInRange` attribute).

Now, all that's left to do is to instantiate our `contactsListView` using our new paginated view, specifying the appropriate collection and "main view" arguments:

assets/js/apps/contacts/list/list_controller.js

```
1  contacts.goTo(1);
2  var contactsListView = new ContactManager.Common.Views.PaginatedView({
3    collection: contacts,
4    mainView: List.Contacts
5  });
```

After all of this work, the pagination controls will display above our main view:



The first page of results, with the pagination controls

Making the Pagination Controls Functional

We need the page to change when a user clicks on the pagination controls. We start by defining which elements can be used to navigate to a different page:

index.html

```
<script type="text/template" id="pagination-controls">
  {{ if(totalPages > 1){ }}
    <ul>
      {{ if(currentPage > 1){ }}
        <li><a href="#">&laquo;</a></li>
        <li><a href="#">&lsaquo;</a></li>
        <li><a href="#" class="navigatable" data-page="1">&laquo;</a></li>
        <li><a href="#" class="navigatable"
               data-page="{{- previous }}">&lsaquo;</a></li>
      {{ }else{ }}
        <li class="disabled"><a href="#">&laquo;</a></li>
        <li class="disabled"><a href="#">&lsaquo;</a></li>
      {{ } }}

      {{ if(pageSet[0] > 1 ){ }}
        <li class="disabled"><a href="#">...</a></li>
      {{ } }}

      {{ _.each(pageSet, function(page){ }}
        {{ if(page === currentPage){ }}
          <li class="active"><a href="#">{{- page }}</a></li>
          <li class="active disabled"><a href="#">{{- page }}</a></li>
        {{ }else{ }}
          <li><a href="#">{{- page }}</a></li>
          <li><a href="#" class="navigatable"
                 data-page="{{- page }}">{{- page }}</a></li>
        {{ } }}
      {{ }); }}

      {{ if(pageSet[pageSet.length - 1] !== lastPage){ }}
        <li class="disabled"><a href="#">...</a></li>
        <li><a href="#">{{- lastPage }}</a></li>
        <li><a href="#" class="navigatable"
               data-page="{{- lastPage }}">{{- lastPage }}</a></li>
      {{ } }}

      {{ if(currentPage !== lastPage){ }}
        <li><a href="#">&rsaquo;</a></li>
        <li><a href="#">&raquo;</a></li>
        <li><a href="#" class="navigatable"
               data-page="{{- next }}">&rsaquo;</a></li>
```

```
42          <li><a href="#" class="navigatable"
43                  data-page="{{- lastPage }}">&raquo;</a></li>
44        {{ }else{ }}
45          <li class="disabled"><a href="#">&rsaquo;</a></li>
46          <li class="disabled"><a href="#">&raquo;</a></li>
47        {{ } }}
48      </ul>
49    {{ } }}
50  </script>
```

As you can see, we've simply added a `navigatable` class to the links that users can click to move to a different page of results, and of course we've added the `page` data attribute to indicate which page should be displayed. With that in place, we can have our pagination controls view respond to the click events and announce them (lines 9-17):

assets/js/common/views.js

```
1   Views.PaginationControls = Marionette.ItemView.extend({
2     template: "#pagination-controls",
3     className: "pagination",
4
5     initialize: function(options){
6       this.paginatedCollection = options.paginatedCollection;
7     },
8
9     events: {
10      "click a[class=navigatable]": "navigateToPage"
11    },
12
13    navigateToPage: function(e){
14      e.preventDefault();
15      var page = parseInt($(e.target).data("page"), 10);
16      this.trigger("page:change", page);
17    },
18
19    serializeData: function(){
20      return this.paginatedCollection.info();
21    }
22  });
```

And since we're encapsulating our paginated view to hide its internal plumbing, the main view will listen for the "page:change" and simply propagate it. That way, wherever our paginated view was created, the event will be available and can be responded to if necessary (lines 7-10):

assets/js/common/views.js

```
Views.PaginatedView = Marionette.LayoutView.extend({
  // edited for brevity

  initialize: function(options){
    // edited for brevity

    var self = this;
    this.listenTo(controls, "page:change", function(page){
      self.trigger("page:change", page);
    });

    this.on("show", function(){
      this.paginationControlsRegion.show(controls);
      this.paginationMainRegion.show(listView);
    });
  }
});
```

Great! What still needs to happen is for our collection to update its current page when the pagination controls are clicked. The way we'll handle this is by using a Backbone model to handle our pagination parameters, which we'll add to our collection when it gets instantiated:

assets/js/entities/contact.js

```
Entities.ContactCollection = Backbone.Paginator.clientPager.extend({
  model: Entities.Contact,

  initialize: function(options){
    options || (options = {});
    var params = options.parameters || { page: 1 };
    this.parameters = new Backbone.Model(params);

    var self = this;
    this.listenTo(this.parameters, "page:change", function(params){
      self.goTo(parseInt(self.parameters.get("page"), 10));
    });
  },

  // edited for brevity
```

As you can see, we make sure our collection has a `parameters` attribute on lines 6-7, and then we update the collection's page when it changes (lines 10-12). With that in place, we simply need to change the `page` parameter as a user navigates (line 7):

assets/js/common/views.js

```
1  Views.PaginationControls = Marionette.ItemView.extend({
2    // edited for brevity
3
4    navigateToPage: function(e){
5      e.preventDefault();
6      var page = parseInt($(e.target).data("page"), 10);
7      this.paginatedCollection.parameters.set("page", page);
8      this.trigger("page:change", page);
9    },
10
11   serializeData: function(){
12     return this.paginatedCollection.info();
13   }
14 });
```

Now, as we click around, we can see the displayed data changes, although our pagination controls aren't entirely functional as they don't update:

- the current page indicator doesn't change, which means the link to page 1 remains unclickable;
- the "next page" link always goes to page 2, even if we are no longer on page 1;
- the range of displayed pages remains the same (when we're on page 5, the range *should* be pages 3-7).

We'll correct that by rerendering the view when the current page changes:

assets/js/common/views.js

```
1  Views.PaginationControls = Marionette.ItemView.extend({
2    template: "#pagination-controls",
3    className: "pagination",
4
5    initialize: function(options){
6      this.paginatedCollection = options.paginatedCollection;
7      this.listenTo(this.paginatedCollection.parameters,
8                                      "page:change", this.render);
9    },
10
11   // edited for brevity
```

If we navigate within our app, the pagination controls still don't update properly... Why? Because the "page:change" event is triggered before the change has actually taken place. That means that the collection's `currentPage` still hasn't changed, and therefore the controls will appear the same. In other words, the pagination controls *do* get updated properly, but they're getting rendered with the same parameters and therefore don't look any different to us. To fix this, we need to trigger an event once the page change has taken place (line 11):

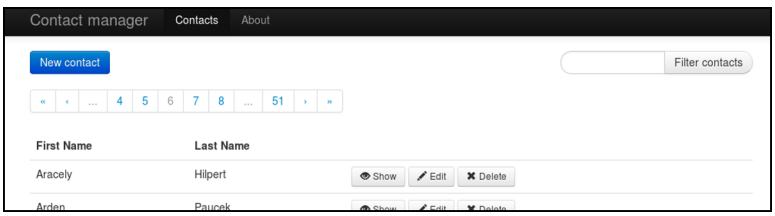
assets/js/entities/contact.js

```
1  Entities.ContactCollection = Backbone.Paginator.clientPager.extend({
2    model: Entities.Contact,
3
4    initialize: function(options){
5      options || (options = {});
6      this.parameters = options.parameters || new Backbone.Model({ page: 1 });
7
8      var self = this;
9      this.listenTo(this.parameters, "page:change", function(params){
10       self.goTo(parseInt(self.parameters.get("page"), 10));
11       self.trigger("page:change:after");
12     });
13   },
14
15   // edited for brevity
```

Now we simply listen for that event in our pagination controls:

assets/js/common/views.js

```
1  initialize: function(options){
2    this.paginatedCollection = options.paginatedCollection;
3    this.listenTo(this.paginatedCollection, "page:change:after", this.render);
4  },
```



The updated pagination controls

Filtering

Let's now restore the filtering functionality we previously ignored, this time using Backbone.paginator's built-in capabilities. First, we set the parameters where appropriate:

assets/js/apps/contacts/list/list_controller.js

```
1  // edited for brevity
2
3  $.when(fetchingContacts).done(function(contacts){
4    if(criterion){
5  ~~    contacts.filter(criterion);~~
6      contacts.parameters.set({ criterion: criterion });
7      contactsListPanel.once("show", function(){
8        contactsListPanel.triggerMethod("set:filter:criterion", criterion);
9      });
10   }
11
12   contacts.goTo(1);
13   var contactsListView = new ContactManager.Common.Views.PaginatedView({
14     collection: contacts,
15     mainView: List.Contacts
16   });
17
18   contactsListPanel.on("contacts:filter", function(filterCriterion){
19 ~~    contacts.filter(criterion);~~
20     contacts.parameters.set({
21       page: 1,
22       criterion: filterCriterion
23     });
24     ContactManager.trigger("contacts:filter", filterCriterion);
25   });
26
27   // edited for brevity
```

Notice that when we filter with a new criterion (lines 20-23), we reset the page number to 1 since we'll be starting the display back on the first page. In contrast, on line 6 we don't set the page number so the user can navigate directly to a filtered results page.

Then we need to react and actually filter the collection when the `criterion` value changes:

assets/js/entities/contact.js

```
Entities.ContactCollection = Backbone.Paginator.clientPager.extend({
  model: Entities.Contact,

  initialize: function(options){
    options || (options = {});
    this.parameters = options.parameters || new Backbone.Model({ page: 1 });

    var self = this;
    this.listenTo(this.parameters, "page:change", function(){
      self.goTo(parseInt(self.parameters.get("page"), 10));
    this.listenTo(this.parameters, "change", function(model){
      if(_.has(model.changed, "criterion")){
        self.setFilter(["firstName", "lastName", "phoneNumber"],
                                  self.parameters.get("criterion"));
      }
      if(_.has(model.changed, "page")){
        self.goTo(parseInt(self.parameters.get("page"), 10));
      }
      self.trigger("page:change:after");
    });
  },

  // edited for brevity
```

We didn't just add a "change:criterion" event listener, because the changes to `criterion` and `page` values aren't fully independent: we need to first filter the collection (if necessary), *then* trigger the page change, so that the pagination controls get udpated with the proper information. Since the filtered collection will contain fewer elements, the number of pages in the controls will need to change (along with resetting the page to 1). Therefore, to check whether a given attribute has changed, we use `_.has` to inspect whether the model's `changed` property (which is an object containing all changed attributes) contains the attribute we're interested in.

Propagating Events

Our pagination works well, but if you click on one of the item view buttons, nothing happens. Why is that? If you look at our list controller, we've got this type of code:

assets/js/apps/contacts/list/list_controller.js

```
var contactsListView = new ContactManager.Common.Views.PaginatedView({
  collection: contacts,
  mainView: List.Contacts
});

// edited for brevity

contactsListView.on("childview:contact:show", function(childView, model){
  ContactManager.trigger("contact:show", model.get("id"));
});
```

The problem with the above code is that we're listening for item view events on the paginated view, but these events don't actually happen on the paginated view (they take place on the view *inside* the paginated view). What can we do? Let's simply give the paginated view a list of events it should propagate. That way we can simply continue listening for the events and everything will work correctly. First, we need to provide the list of events we're interested in:

assets/js/apps/contacts/list/list_controller.js

```
contacts.goTo(1);
var contactsListView = new ContactManager.Common.Views.PaginatedView({
  collection: contacts,
  mainView: List.Contacts,
  propagatedEvents: [
    "childview:contact:show",
    "childview:contact:edit",
    "childview:contact:delete"
  ]
});
```

And in our paginated view, we simply need to register event listeners for the events that need to be propagated, and manually retrigger the event:

assets/js/common/views.js

```
Views.PaginatedView = Marionette.LayoutView.extend({
  // edited for brevity

  initialize: function(options){
    this.collection = options.collection;
    var eventsToPropagate = options.propagatedEvents || [];

    // edited for brevity

    var self = this;
    this.listenTo(controls, "page:change", function(page){
      self.trigger("page:change", page);
    });
    _.each(eventsToPropagate, function(evt){
      self.listenTo(listView, evt, function(view, model){
        self.trigger(evt, view, model);
      });
    });

    // edited for brevity
  }
});
```

Now, if (e.g.) a "childview:contact:show" event is triggered in the composite view within the paginated view, the latter will retrigger that same event. Therefore, our list controller can simply continue listening for the "childview:contact:show" event and react as before.

Keeping the URL in Sync

Let's now indicate the current display criteria in the URL so that users can bookmark pages and come back later. For starters, we need our "list" controller to propagate the "page:change" event so it can be handled in our sub-app manager:

assets/js/apps/contacts/list/list_controller.js

```
1  var contactsListView = new ContactManager.Common.Views.PaginatedView({
2    // edited for brevity
3  });
4
5  contactsListView.on("page:change", function(){
6    ContactManager.trigger("page:change", _.clone(contacts.parameters.attributes));
7  });
8
9  contactsListPanel.on("contacts:filter", function(filterCriterion){
10   // edited for brevity
```

Notice that we once again `clone` the model's attributes on line 6 so that if they get modified, our original model won't be affected.

And now, we need to listen for this event and update the URL, which will mean serializing the current collection parameters (page, filter criterion, etc.) to add them to the URL:

assets/js/apps/contacts/contacts_app.js

```
1  ContactManager.on("page:change", function(options){
2    ContactManager.navigate("contacts/filter/" + serializeParams(options));
3  });
4
5  ContactManager.on("contacts:list", function(){
6    // edited for brevity
```

We'll use the "contacts/filter" URL when we have parameters, because it will be easy for us to differentiate when implementing routing rules. If you remember, we already have a route for "contacts/:id" so we can't simply define a "contacts/:parameters" route, or both of them would conflict.

So what will this `serializeParams` function do? It needs to

1. select only the attributes we want to display in the URL (i.e. page, and filter criterion);
2. remove the attributes with no values: if the filter criterion is empty (because it's been erased), the "criterion" attribute shouldn't be in the URL;
3. join each key-value pair with a ":", so we'd obtain (e.g.) "page:3";
4. join all the pairs together with a "+", so we'd end up with (e.g.) "page:3+criterion:ck".

Starting with Backbone 1.1.0, query strings are completely ignored by Backbone (see discussion here[96] and here[97], preventing us from using query strings to pass the argument around. In other words, it isn't possible to use a URL such as "#contacts?filter=...". We'll therefore be using the format suggested by Jeremy Ashkenas (Backbone's author).

That said, Backbone 1.1.1 has introcued an `execute` method in the router, which *can* be used to process query strings. We'll cover that later (TODO: link).

Now that you know what's going on, let's use some of Underscore's methods to do the work for us:

assets/js/apps/contacts/contacts_app.js

```
ContactsApp.Router = Marionette.AppRouter.extend({
  appRoutes: {
    // edited for brevity
  }
});

var serializeParams = function(options){
  options = _.pick(options, "criterion", "page");
  return (_.map(_.filter(_.pairs(options), function(pair){ return pair[1]; }),
                         function(pair){ return pair.join(":"); })).join("+");
};

var API = {
  // edited for brevity
```

If we now got to our contacts index page and click on a direct link to a page (in the pagination controls bar), we can see that the URL updates itself to indicate the "page" attribute. Let's implement the same for our filter criterion:

assets/js/apps/contacts/list/list_controller.js

```
contactsListPanel.on("contacts:filter", function(filterCriterion){
  contacts.parameters.set({
    page: 1,
    criterion: filterCriterion
  });
  ContactManager.trigger("contacts:filter", filterCriterion);
  ContactManager.trigger("contacts:filter",
                    _.clone(contacts.parameters.attributes));
});
```

[96] https://github.com/jashkenas/backbone/issues/2801

[97] https://github.com/jashkenas/backbone/issues/891

Naturally, we need to respond to the filter event and serialize the parameters:

assets/js/apps/contacts_app.js

```
ContactManager.on("contacts:filter", function(criterion){
  if(criterion){
    ContactManager.navigate("contacts/filter/criterion:" + criterion);
  }
  else{
    ContactManager.navigate("contacts");
  }
ContactManager.on("contacts:filter", function(options){
  ContactManager.navigate("contacts/filter/" + serializeParams(options));
});
```

Now when the user clicks around, we can see the URL updating, including the filter criterion. In addition, we can see that if we navigate to a given page, then filter the contacts, the page parameter will be reset to 1.

Loading State from the URL

Since our app updates the URL paramters as the internal state evolves, what we still need to is to "jump start" the app's internal state according to the URL parameters (in case our user comes to the app through a bookmark, for example).

The first thing we need is to define a route:

assets/js/apps/contacts/contacts_app.js

```
ContactsApp.Router = Marionette.AppRouter.extend({
  appRoutes: {
    "contacts(/filter/criterion::criterion)": "listContacts",
    "contacts(/filter/:params)": "listContacts",
    "contacts/:id": "showContact",
    "contacts/:id/edit": "editContact"
  }
});
```

Then, we'll need a function to parse the URL parameters into a usable object, in effect reversing the `serializeParams` we defined earlier:

assets/js/apps/contacts/contacts_app.js

```
1  var parseParams = function(params){
2    var options = {};
3    if(params && params.trim() !== ''){
4      params = params.split('+');
5      _.each(params, function(param){
6        var values = param.split(':');
7        if(values[1]){
8          if(values[0] === "page"){
9            options[values[0]] = parseInt(values[1], 10);
10         }
11         else{
12           options[values[0]] = values[1];
13         }
14       }
15     });
16   }
17   _.defaults(options, { page: 1 });
18   return options;
19 };
20
21 var serializeParams = function(options){
22   // edited for brevity
```

Here's what's going on:

1. we start by defining an empty object, to cover cases where no parameters are found in the URL (line 2);
2. on line 3, we check for the presence of parameters;
3. we then need to split our string on line 4 at the "+" separator, to obtain our parameter pairs;
4. then (lines 5-15), we
    1. iterate over these pairs and split the key-value pair along the ":" separator (line 6);
    2. on lines 8-13, add the value to the "key" attribute on the `options` object (and parse the page number if appropriate), but only if there is a non-empty value (line 7);
5. if no page number was provided in the URL, use 1 as the default value (line 17);
6. finally, we return our parsed `options` object on line 18.

Then, we simply need to parse the URL params provided to us by the router:

assets/js/apps/contacts/contacts_app.js

```
var API = {
 listContacts: function(criterion){
   ContactsApp.List.Controller.listContacts(criterion);
 listContacts: function(params){
   var options = parseParams(params);
   ContactsApp.List.Controller.listContacts(options);
   ContactManager.execute("set:active:header", "contacts");
 },
```

And since we're now providing `Controller.listContacts` with an `options` object instead of a `criterion` value, let's go ahead and update it:

assets/js/apps/contacts/list/list_controller.js

```
List.Controller = {
 listContacts: function(criterion){
  listContacts: function(options){
    // edited for brevity

    $.when(fetchingContacts).done(function(contacts){
      if(criterion){
      if(options.criterion){
        contacts.parameters.set({
          page: 1,
          criterion: criterion
          criterion: options.criterion
        });
        contactsListPanel.once("show", function(){
          contactsListPanel.triggerMethod("set:filter:criterion", criterion);
          contactsListPanel.triggerMethod("set:filter:criterion",
                                                    options.criterion);
        });
      }

      contacts.goTo(1);
      contacts.goTo(options.page);
      var contactsListView = new ContactManager.Common.Views.PaginatedView({
        collection: contacts,
        mainView: List.Contacts
      });
```

```
27
28      // edited for brevity
```

Git commit implementing a basic paginated view

3b165449412fb8f8b19c2695efe1ee99f6b54355[98]

Using Proper `href` Attributes

Although our pagination controls behave properly and update the main view, we still need to use correct `href` attributes for our pagination links. Since these will change depending on the collection, where it is displayed, etc. we will have to provide an extra parameter when we initialize our paginated view:

assets/js/apps/contacts/list/list_controller.js

```
1  var contactsListView = new ContactManager.Common.Views.PaginatedView({
2    collection: contacts,
3    mainView: List.Contacts,
4    paginatedUrlBase: "contacts/filter/"
5  });
```

We then need to provide this attribute to the pagination controls (line 9):

assets/js/common/views.js

```
1  Views.PaginatedView = Marionette.LayoutView.extend({
2    // edited for brevity
3
4    initialize: function(options){
5      this.collection = options.collection;
6
7      var controls = new Views.PaginationControls({
8        paginatedCollection: this.collection,
9        urlBase: options.paginatedUrlBase
10     });
```

Of course, this value needs to be stored in our pagination controls view (line 7):

[98] https://github.com/davidsulc/marionette-serious-progression-3b165449412fb8f8b19c2695efe1ee99f6b54355

assets/js/common/views.js

```
Views.PaginationControls = Marionette.ItemView.extend({
  template: "#pagination-controls",
  className: "pagination",

  initialize: function(options){
    this.paginatedCollection = options.paginatedCollection;
    this.urlBase = options.urlBase;
    this.listenTo(this.paginatedCollection, "page:change:after", this.render);
  },
```

Finally, we need to provide the template with this value:

assets/js/common/views.js

```
Views.PaginationControls = Marionette.ItemView.extend({
  // edited for brevity

  serializeData: function(){
    var data = this.paginatedCollection.info(),
        url = this.urlBase,
        criterion = this.paginatedCollection.parameters.get("criterion");
    if(criterion){
      url += "criterion:" + criterion + "+";
    }
    url += "page:";
    data.urlBase = url;
    return data;
  }
});
```

And now, we can use the `urlBase` attribute to define proper `href` values:

index.html

```
<script type="text/template" id="pagination-controls">
  {{ if(totalPages > 1){ }}
    <ul>
      {{ if(currentPage > 1){ }}
        <li><a href="#" class="navigatable" data-page="1">&laquo;</a></li>
        <li><a href="#" class="navigatable"
                              data-page="{{- previous }}">&lsaquo;</a></li>
        <li><a href="#{{- urlBase + 1 }}" class="navigatable"
                                          data-page="1">&laquo;</a></li>
        <li><a href="#{{- urlBase + previous }}" class="navigatable"
                               data-page="{{- previous }}">&lsaquo;</a></li>
      {{ }else{ }}
        <li class="disabled"><a href="#">&laquo;</a></li>
        <li class="disabled"><a href="#">&lsaquo;</a></li>
      {{ } }}

      {{ if(pageSet[0] > 1 ){ }}
        <li class="disabled"><a href="#">...</a></li>
      {{ } }}

      {{ _.each(pageSet, function(page){ }}
        {{ if(page === currentPage){ }}
          <li class="active disabled"><a href="#">{{- page }}</a></li>
        {{ }else{ }}
          <li><a href="#" class="navigatable"
                             data-page="{{- page }}">{{- page }}</a></li>
          <li><a href="#{{- urlBase + page }}" class="navigatable"
                            data-page="{{- page }}">{{- page }}</a></li>
        {{ } }}
      {{ }); }}

      {{ if(pageSet[pageSet.length - 1] !== lastPage){ }}
        <li class="disabled"><a href="#">...</a></li>
        <li><a href="#" class="navigatable"
                      data-page="{{- lastPage }}">{{- lastPage }}</a></li>
        <li><a href="#{{- urlBase + lastPage }}" class="navigatable"
                       data-page="{{- lastPage }}">{{- lastPage }}</a></li>
      {{ } }}

      {{ if(currentPage !== lastPage){ }}
        <li><a href="#" class="navigatable"
```

```
                        data-page="{{- next }}">&rsaquo;</a></li>
        <li><a href="#" class="navigatable"
                        data-page="{{- lastPage }}">&raquo;</a></li>
        <li><a href="#{{- urlBase + next }}" class="navigatable"
                          data-page="{{- next }}">&rsaquo;</a></li>
        <li><a href="#{{- urlBase + lastPage }}" class="navigatable"
                        data-page="{{- lastPage }}">&raquo;</a></li>
      {{ }else{ }}
        <li class="disabled"><a href="#">&rsaquo;</a></li>
        <li class="disabled"><a href="#">&raquo;</a></li>
      {{ } }}
    </ul>
  {{ } }}
</script>
```

There! Our pagination links now have the correct `href` attribute, so if they're opened in a new tab they will behave correctly.

Using Multiple Models in an ItemView

Let's take a brief moment to look deeper into how we used `serializeData` above to merge multiple attributes and make them available to the view. This technique can be quite useful, for example when you need to display multiple models in a single item view.

In the code above, we're creating a `data` object containing the attributes we want to provide to the template. And, as we already know, we can provide arbitrary values to views when they're initialized which will be available in the `options` object. So let's take all of this together and use 2 model in a single item view...

The code below is for explanation purposes only, and won't get included in the code base.

Create a template we'll use

```
<script type="text/template" id="relationship-display">
    <p>{{- assistant.firstName }} has been {{- boss.firstName }}'s assitant
       for {{- duration }} years.</p>
</script>
```

Define the view we'll use

```
var RelationshipView = Backbone.Marionette.ItemView.extend({
    template: "#relationship-display",
    initialize: function(options){
      this.boss = options.boss;
      this.assistant = options.assistant;
      this.relationchipDuration = options.relationshipDuration;
    },
    serializeData: function(){
      var data = {};
      data.boss = _.clone(this.boss.attributes);
      data.assistant = _.clone(this.assistant.attributes);
      data.duration = this.relationshipDuration;
      return data;
    }
  });
```

Instantiate the data we'll need

```
var alice = new ContactManager.Entities.Contact({
    firstName: "Alice",
    lastName: "Arten"
  });

var bob = new ContactManager.Entities.Contact({
    firstName: "Bob",
    lastName: "Brigham"
  });

var view = new RelationshipView({
    boss: alice,
    assistant: bob,
    relationshipDuration: 4
  });

ContactManager.mainRegion.show(view);
```

As you can see, we can freely define the data that is made available to the template from within `serializeData`: we can use objects, rename attributes, and so on. Then, within the template, we simply access the attributes provided by `serializeData`, even if they are objects themselves, such as `boss.firstName`.

Git commit managing URLs for paginated view
c56169d47889283488ba06eb1da172192e5ef7d2[99]

Using RequestPager

API Properties

We'll use a "contacts_paginated" endpoint to provide us with paginated results of contacts. It will accept 3 parameters:

- `count`: how many results should be returned (maximum);
- `offset`: how many results should be skipped before returning data (similar to the SQL offset parameter);
- `filter`: a filter criterion the result set must conform to.

The result set will be contained in a `results` attribute, and a `resultCount` attribute will indicate the total number of results for the received query:

```
{
  "resultCount":503,
  "results":[
    {
      "id":1901, "firstName":"Abdul", "lastName":"Dicki",
      "phoneNumber":"1-600-307-5332",
      "url":"http://localhost:3000/contacts/1901.json"
    },
    {
      "id":1928,"firstName":"Abelardo","lastName":"Bernier",
      "phoneNumber":"1-082-345-8785 x03541",
      "url":"http://localhost:3000/contacts/1928.json"
    },
    // other contacts
    {
      "id":1921,"firstName":"Ahmad","lastName":"Heaney",
      "phoneNumber":"(754)081-5369",
      "url":"http://localhost:3000/contacts/1921.json"
    }
  ]
}
```

[99] https://github.com/davidsulc/marionette-serious-progression-c56169d47889283488ba06eb1da172192e5ef7d2

Given the same filter criterion (but a different page request), the content of the `results` attribute will change but the `resultsCount` value will remain constant. If the filter criterion changes, it is likely the `resultCount` will vary also.

Adapting the Application

We'll start by setting up our request page properly:

assets/js/entities/contact.js

```
~~Entities.ContactCollection = Backbone.Paginator.clientPager.extend({~~
Entities.ContactCollection = Backbone.Paginator.requestPager.extend({
  model: Entities.Contact,

  // edited for brevity

  paginator_core: {
    dataType: "json",
~~    url: "contacts"~~
    url: "contacts_paginated?"
  },
  paginator_ui: {
    firstPage: 1,
    currentPage: 1,
    perPage: 10,
    pagesInRange: 2
  },
  server_api: {
    count: function() { return this.perPage },
    offset: function() {
      return ((this.parameters.get("page") || 1) - 1) * this.perPage
    },
    filter: function() { return this.parameters.get("criterion"); }
  },
  parse: function (response) {
    var data = response.results;
    this.totalRecords = response.resultCount;
    this.totalPages = Math.ceil(this.totalRecords / this.perPage);
    this.currentPage = this.parameters.get("page");
    return data;
  }
```

Let's review our main changes:

- change the collection to be a `requestPager` instead of a `clientPager` (lines 1-2);
- change the API enpoint to "contacts_paginated?" (lines 9-10): notice we include the "?" because it will be followed by query parameters sent to the server (for filtering, pagination, etc.);
- configure the `server_api` object to provide the proper query parameters to the server, according to the collection state (lines 18-24);
- implement the `parse` function to return the collection data, and update collection information according to the server response (lines 25-31).

With the basics in place, we still need to react to collection parameter changes; except this time we'll need to actually fetch the new result set from the server:

assets/js/entities/contact.js

```
1  Entities.ContactCollection = Backbone.Paginator.requestPager.extend({
2    model: Entities.Contact,
3
4    initialize: function(models, options){
5      options || (options = {});
6      var params = options.parameters || { page: 1 };
7      this.parameters = new Backbone.Model(params);
8
9      var self = this;
10     this.listenTo(this.parameters, "change", function(model){
11       if(_.has(model.changed, "criterion")){
12 ~~         self.setFilter(["firstName", "lastName", "phoneNumber"],~~
13 ~~                                    self.parameters.get("criterion"));~~
14         self.server_api.filter = self.parameters.get("criterion");
15       }
16 ~~       if(_.has(model.changed, "page")){~~
17 ~~         self.goTo(parseInt(self.parameters.get("page"), 10));~~
18 ~~       }~~
19 ~~       self.trigger("page:change:after");~~
20       $.when(this.pager()).done(function(){
21         self.trigger("page:change:after");
22       });
23     });
24   },
```

As you can tell, we needed to change our logic slightly:

- if the filter criterion changes, we simply update the `server_api` object;
- each time the `parameters` model changes, a request is fired off to the server (by calling `this.pager()`) to obtain the new data set;
- when the result set is returned, we fire the "page:change:after" so that our pagination controls update correctly.

Note that we need to wait for the `pager()` call to be `done` before triggering the "page:change:after" event: otherwise, the event could get triggered before the data is updated. In that case, the `currentPage` value (among others) won't have been updated yet, and the pagination controls will look exactly the same when they rerender (i.e. they won't display the correct current page).

We still need to initialize our collection with the correct parameters, in order for the correct page of results to be fetched initially:

assets/js/apps/contacts/list/list_controller.js

```
var fetchingContacts = ContactManager.request("contact:entities");
var fetchingContacts = ContactManager.request("contact:entities",
                                              { parameters: options });
```

Since we're querying the proper contacts by providing the `parameters` option, we no longer need to explicitly set the criterion (it's already been done), or to force a page display (since the correct page of results will be fetched from the API):

assets/js/apps/contacts/list/list_controller.js

```
$.when(fetchingContacts).done(function(contacts){
  if(options.criterion){
    contacts.parameters.set({ criterion: criterion });
    contactsListPanel.once("show", function(){
      contactsListPanel.triggerMethod("set:filter:criterion", options.criterion);
    });
  }

  contacts.goTo(options.page);
  var contactsListView = new ContactManager.Common.Views.PaginatedView({

  // edited for brevity
```

Finally, we need our request handler to initialize our collection with the provided parameters:

assets/js/entities/contact.js

```
1  var API = {
2    getContactEntities: function(options){
3      var contacts = new Entities.ContactCollection();
4      var contacts = new Entities.ContactCollection([],
5                          { parameters: options.parameters });
6      delete options.parameters;
7      var defer = $.Deferred();
```

On lines 4-5, we simply provide the parameters to the collection initializer (the first argument is the list of models to initialize the collection with). And since we then no longer need the `parameters` attribute, we delete it on line 6.

Although we've changed our code from client-side pagination to server-side pagination, our "paginated view" hasn't had to change at all. In addition, by managing our navigation through events, we haven't had to alter that code either.

A Small Bugfix

When we navigate to a page, the result set isn't displayed according to sorting order. So each time our request pager syncs with the server, we'll simply sort the collection (silently) and trigger a "reset" event so our composite view rerenders the child views:

assets/js/entities/contact.js

```
1  Entities.ContactCollection = Backbone.Paginator.requestPager.extend({
2    model: Entities.Contact,
3
4    initialize: function(models, options){
5      // edited for brevity
6
7      this.on("sync", function(){
8        this.sort({silent: true});
9        this.trigger("reset");
10     });
11   },
```

Git commit using `requestPager`

c5b870a08f0c8786caaeb665100b4ee1b6bb1161[100]

[100] https://github.com/davidsulc/marionette-serious-progression-c5b870a08f0c8786caaeb665100b4ee1b6bb1161

Memory Management

In this chapter, we'll cover a few topics that will improve your app's performance by reducing its memory footprint.

Defining Methods on Prototypes

In javascript, object instances "inherit" behavior from a meta-object called the prototype. But before we go any further, let's clarify something: speaking of inheritance in this case is merely a handy, but *incorrect*, intellectual shortcut. In effect, object instances have a prototype meta object on which methods will be searched for if they're missing on the object instance itself:

```
1  ContactManager.Entities.Contact.prototype.announce = function(){
2    console.log("I am " + this.get("firstName") + " " +
3                this.get("lastName") + ".");
4  }
5
6  var john = new ContactManager.Entities.Contact({
7    firstName: "John",
8    lastName: "Doe"
9  });
10
11 john.announce(); // I am John Doe.
```

On line 11, we call the `announce` method on `john` even though the `Contact` doesn't define that function. Since the method can't be found on the object itself, the prototype chain is inspected and, sure enough, an `announce` implementation is found and invoked.

Within the function definition in the prototype, we're able to use `this` since it will evaluate to the invoking object: in our case, `john`.

Naturally, we could define a special `announce` implementation for John, by shadowing the `announce` method:

```
ContactManager.Entities.Contact.prototype.announce = function(){
  console.log("I am " + this.get("firstName") + " " +
              this.get("lastName") + ".");
}

var john = new ContactManager.Entities.Contact({
  firstName: "John",
  lastName: "Doe"
});

john.announce = function(){
  console.log("Hi, my name is " + this.get("firstName") +
              " " + this.get("lastName") + ".");
}

john.announce(); // Hi, my name is John Doe.
```

One thing to be well aware of is that the prototype is a shared instance, and function implementations (as well as values) do **not** get copied into "inheriting" object instances: instead, their value is looked up each time.

```
ContactManager.Entities.Contact.prototype.announce = function(){
  console.log("I am " + this.get("firstName") + " " +
              this.get("lastName") + ".");
}

var john = new ContactManager.Entities.Contact({
  firstName: "John",
  lastName: "Doe"
});

ContactManager.Entities.Contact.prototype.announce = function(){
  console.log("Pleased to meet you, I am " + this.get("firstName") +
              " " + this.get("lastName") + ".");
}

var jane = new ContactManager.Entities.Contact({
  firstName: "Jane",
  lastName: "Doe"
});

john.announce(); // Pleased to meet you, I am John Doe.
jane.announce(); // Pleased to meet you, I am Jane Doe.
```

In this example, we create the `john` instance before changing the `announce` implementation. But as you can see, when we later execute the method the result is the same for both `john` and `jane`. That is because, when we call `announce` on lines 21-22, the implementation defined on the prototype is the one we defined on lines 11-14.

Now, let's say that John and Jane *aren't* married to each other, and each have their own kids. Let's try keeping track of the kids using the prototype:

```
1  ContactManager.Entities.Contact.prototype.kids = [];
2
3  var john = new ContactManager.Entities.Contact({
4    firstName: "John",
5    lastName: "Doe"
6  });
7  var jane = new ContactManager.Entities.Contact({
8    firstName: "Jane",
9    lastName: "Doe"
10 });
11
12 john.kids.push("Mike");
13 jane.kids.push("Noah");
14
15 console.log(john.kids); // ["Mike", "Noah"]
16 console.log(jane.kids); // ["Mike", "Noah"]
```

Uh oh! We wanted to keep the list of kids separate, but since it's shared through the prototype, anytime we change the `kids` value it is reflected across all other contacts.

They key take away is that the prototype object is shared, and functions/values do **not** get copied over to "child" instances, which means:

- defining common functions on the prototype reduces memory usage, since the implementation will exist only once, instead of being created/copied for each object instance;
- internal state variables should be declared on the object instances, because you don't want them shared through the prototype.

With that bit of theory having been cleared up, let's move generic values and functions into their respective prototypes!

We can start with the common form view in file *assets/js/apps/contacts/common/views.js*. Here's the current code:

assets/js/apps/contacts/common/view.js

```
Views.Form = Marionette.ItemView.extend({
  template: "#contact-form",

  events: {
    "click button.js-submit": "submitClicked"
  },

  submitClicked: function(e){
    e.preventDefault();
    var data = Backbone.Syphon.serialize(this);
    this.trigger("form:submit", data);
  },

  onFormDataInvalid: function(errors){
    var $view = this.$el;

    var clearFormErrors = function(){
      var $form = $view.find("form");
      $form.find(".help-inline.error").each(function(){
        $(this).remove();
      });
      $form.find(".control-group.error").each(function(){
        $(this).removeClass("error");
      });
    }

    var markErrors = function(value, key){
      var $controlGroup = $view.find("#contact-" + key).parent();
      var $errorEl = $("<span>", { class: "help-inline error", text: value });
      $controlGroup.append($errorEl).addClass("error");
    }

    clearFormErrors();
    _.each(errors, markErrors);
  }
});
```

The `template` and `events` attributes need to be linked to the individual view: changing one form's `template` attribute shouldn't affect the other forms! On the other hand, `submitClicked` and `onFormDataInvalid` are "standard" behavior implementations and can therefore be moved to the prototype:

assets/js/apps/contacts/common/view.js

```
Views.Form = Marionette.ItemView.extend({
  template: "#contact-form",

  events: {
    "click button.js-submit": "submitClicked"
  }
});

_.extend(View.Form.prototype, {
  submitClicked: function(e){
    e.preventDefault();
    var data = Backbone.Syphon.serialize(this);
    this.trigger("form:submit", data);
  },

  onFormDataInvalid: function(errors){
    var $view = this.$el;

    var clearFormErrors = function(){
      var $form = $view.find("form");
      $form.find(".help-inline.error").each(function(){
        $(this).remove();
      });
      $form.find(".control-group.error").each(function(){
        $(this).removeClass("error");
      });
    }

    var markErrors = function(value, key){
      var $controlGroup = $view.find("#contact-" + key).parent();
      var $errorEl = $("<span>", { class: "help-inline error", text: value });
      $controlGroup.append($errorEl).addClass("error");
    }

    clearFormErrors();
    _.each(errors, markErrors);
  }
});
```

As an exercise, go through the various view files in the project and refactor them to move the appropriate function definitions to the respective prototypes. Once you're done, take a look at the

Git commit referenced at the end of this section.

With the views taken care of, let's move on to improving our model and collection entities, starting with our contacts:

assets/js/entities/contact.js

```
Entities.Contact = Entities.BaseModel.extend({
  urlRoot: "contacts",

  initialize: function(){
    // edited for brevity
  },

  defaults: {
    // edited for brevity
  }
});

_.extend(Entities.Contact.prototype, Backbone.Validation.mixin, {
  parse: function(response){
    // edited for brevity
  },

  validation: {
    // edited for brevity
  },

  sync: function(method, model, options){
    // edited for brevity
  }
});

Entities.ContactCollection = Backbone.Paginator.requestPager.extend({
  model: Entities.Contact,

  initialize: function(models, options){
    // edited for brevity
  },

  comparator: "firstName",
  paginator_core: {
    // edited for brevity
  },
```

```
38    paginator_ui: {
39      // edited for brevity
40    },
41    server_api: {
42      // edited for brevity
43    }
44  });
45
46  _.extend(Entities.ContactCollection.prototype, {
47    parse: function (response) {
48      var data = response.results;
49      this.totalRecords = response.resultCount;
50      this.totalPages = Math.ceil(this.totalRecords / this.perPage);
51      this.currentPage = this.parameters.get("page");
52      return data;
53    }
54  });
```

On line 13, we're simply adding an object by which we wish to extend the prototype: we were already extending it, so we can add on arguments in the same call. The code is equivalent to

```
1  _.extend(Entities.Contact.prototype, Backbone.Validation.mixin);
2  _.extend(Entities.Contact.prototype, {
3    // the object with which to extend the prototype, as above
4  });
```

As an exercise, try to modify the "repository" entity to better leverage the prototype before proceeding.

Our modified repository entity file becomes:

assets/js/entities/contact.js

```
Entities.Repository = Backbone.Model.extend({
  initialize: function(options){
    // edited for brevity
  },

  urlRoot: "https://api.github.com"
});

_.extend(Entities.Repository.prototype, {
  url: function(){
    return _.result(this, "urlRoot") + "/repos/" + this.username + "/" +
                                  (this.get("githubName") || this.get("name"));
  },

  validate: function(attrs, options) {
    var errors = {}
    if (! attrs.name) {
      errors.name = "can't be blank";
    }
    if( ! _.isEmpty(errors)){
      return errors;
    }
  },

  sync: function(method, model, options){
    // edited for brevity
  }
});
```

Git commit reducing the app's memory footprint by leveraging prototypes:

5bf07fdec0c8ba635e175d1aad367484e4e8dd9b[101]

There's a bug in the code above: the pagination configuration should be stored within the contact collection instance (as opposed to a "class property"), as we'll demonstrate (and fix!) in chapter Model Relationships.

[101] https://github.com/davidsulc/marionette-serious-progression-5bf07fdec0c8ba635e175d1aad367484e4e8dd9b

Using `listenTo` Instead of `on`

As you've probably noticed, as we build our app we often need to react to events happening elsewhere in the application, and we've done so in two ways:

Using `on` (assets/js/apps/contacts/list/list_controller.js)

```
1 contactsListView.on("page:change", function(){
2   ContactManager.trigger("page:change",
3                          _.clone(contacts.parameters.attributes));
4 });
```

Using `listenTo` (assets/js/apps/config/marionette/regions/dialog.js)

```
1 onShow: function(view){
2   this.listenTo(view, "dialog:close", this.closeDialog);
3
4   // edited for brevity
5
6   this.listenTo(view, "render", configureDialog);
7 },
```

Why do we have 2 separate ways to do the same thing? The simple explanation is that using `on` is the "traditional" way of registering callbacks to react to a certain event taking place. By using `listenTo`, you achieve the same result, with the following advantage (from the documentation[102]):

> The advantage of using this form, instead of `other.on(event, callback, object)`, is that `listenTo` allows the object to keep track of the events, and they can be removed all at once later on.

Why is being able to remove all event listeners at once important? Well, since we're in the "memory management" chapter, you're probably guessing it has something to do with memory.

Javascript uses a mark-and-sweep garbage-collector to free up memory as objects are no longer needed. But how does one decide that an object is no longer needed? You start at the root object (which is `window` in browsers), find all objects referenced from the root object, then all objects referenced from these, and so on. In other words, the garbage collector will find all *reachable* objects and collect all non-reachable objects (i.e. free up their memory). For more on garbage collection, take a look at this MDN article[103] from which this explanation has been adapted.

Now that you're somewhat familiar with how memory gets freed by the javascript garbage collector, let's take a moment to decribe the issue with event callbacks.

Let's say we had the following code for the view listing our contacts:

[102] http://http://www.backbonejs.org#Events-listenTo

[103] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management

assets/js/apps/contacts/list/list_view.js

```
List.Contacts = Marionette.CompositeView.extend({
  // edited for brevity

  initialize: function(){
    var self = this;
    this.collection.on("reset", function(){
      self.attachHtml = function(collectionView, childView, index){
        collectionView.$el.append(childView.el);
      }
    });
  }
});
```

The problem here is that once we no longer need the view instance, it can't get garbage collected: the `collection` instance holds a reference to the view instance, which makes the view instance reachable. And as we discussed above: as long as an object is reachable it can't get garbage collected so its memory isn't freed.

Let's use an analogy to better understand memory freeing: sharing your books. Let's say that each time your lend a book to a friend, you rely on him to return it once he's done with it. In this case, it will be difficult to ask your friends to return all your books when you need them, because you can't remember whom you lent books to, and you didn't keep track of it either...

What if instead, each time you lent a book, you wrote down the book's title and the friend's name? Then, anytime you need to have all of your books returned, you simply look at the list, request that all your books be returned, and cross the books off the list as they are returned to you.

In the above examples, the former would be using `on` and the latter `listenTo`. This is because Backbone now implements a `stopListening`[104] method that can be called without arguments to make it stop listening to all events registered using `listenTo`. And since views call `stopListening` before being closed and removed, the code above is better written like so (as it is in our app):

[104] http://www.backbonejs.org#Events-stopListening

assets/js/apps/contacts/list/list_view.js

```
List.Contacts = Marionette.CompositeView.extend({
  // edited for brevity

  initialize: function(){
    this.listenTo(this.collection, "reset", function(){
      this.attachHtml = function(collectionView, childView, index){
        collectionView.$el.append(childView.el);
      }
    });
  }
});
```

When this view is no longer needed, `stopListening` will be called and the view instance will be able to be garbage collected. Does this mean that any time you use `on` to register an event handler you're creating a memory leak? No, but to keep things simple, try to always use `listenTo` with the following exceptions for using `on`:

- calling it on `this` with no other object instances referenced in the callback (i.e. using only `this`):

assets/js/entities/contact.js

```
this.on("change", function(){
  this.set("fullName", this.get("firstName") +
           " " + this.get("lastName"));
});
```

- you're not within a Backbone object (e.g. the current context is `window`) and need to register a listener on your app:

assets/js/app.js

```
ContactManager.on("start", function(){
  _.templateSettings = {
    interpolate: /\{\{=(.+?)\}\}/g,
    escape: /\{\{-(.+?)\}\}/g,
    evaluate: /\{\{(.+?)\}\}/g
  };
});
```

Before we update our app to use `listenTo` wherever possible, let's see how Marionette controllers fit in the picture.

Using Marionette Controllers

By using a Marionette `Controller` object instead of a plain javascript object, we gain access to an event binder and can therefore use `listenTo` within our controllers. Then, when our controllers are closed (e.g. when they are stopped as explained in the next section) `stopListening` will be called automatically.

Let's start by converting our "headers" app list controller. Here's the original code:

assets/js/apps/header/list/list_controller.js

```
List.Controller = {
  listHeader: function(){
    // edited for brevity
  },

  setActiveHeader: function(headerUrl){
    // edited for brevity
  }
};
```

To convert this to a Marionette `Controller`, we'll simply use a private-scoped variable to store the controller definition, and then attach the controller instance to the public attribute:

assets/js/apps/header/list/list_controller.js

```
var Controller = Marionette.Controller.extend({
  listHeader: function(){
    // edited for brevity
  },

  setActiveHeader: function(headerUrl){
    // edited for brevity
  }
});

List.Controller = new Controller();
```

But since we now have a controller object, we can also convert our `on` event bindings to `listenTo`:

assets/js/apps/header/list/list_controller.js

```
var Controller = Marionette.Controller.extend({
  listHeader: function(){
    var links = ContactManager.request("header:entities");
    var headers = new List.Headers({collection: links});

    headers.on("brand:clicked", function(){
    this.listenTo(headers, "brand:clicked", function(){
      ContactManager.trigger("contacts:list");
    });

    headers.on("childview:navigate", function(childView, model){
    this.listenTo(headers, "childview:navigate", function(childView, model){
      var trigger = model.get("navigationTrigger");
      ContactManager.trigger(trigger);
    });

    ContactManager.headerRegion.show(headers);
  },

  setActiveHeader: function(headerUrl){
    var links = ContactManager.request("header:entities");
    var headerToSelect = links.find(function(header){
      return header.get("url") === headerUrl;
    });
    headerToSelect.select();
    links.trigger("reset");
  }
});

List.Controller = new Controller();
```

Try to convert all controller objects to Marionette `Controller` instances, and replace `on` event bindings with `listenTo` where possible. Then, compare with the following commit.

Git commit using Marionette controllers and `listenTo`:

40a15de87db9bfa9faa23279895fb431962919ff[105]

[105] https://github.com/davidsulc/marionette-serious-progression-40a15de87db9bfa9faa23279895fb431962919ff

Module Start/Stop

(If you've read my book on Marionette and RequireJS[106], you'll notice the content in this section is very similar to the chapter on the same subject in the RequireJS book. You can probably safely skip most of this section, as the main difference is we've converted to using Marionette controllers and `listenTo`, but the start/stop logic remains the same. I do suggest you take a look at the end of the chapter, to see how the controllers are stopped when the sub-app is stopped.)

In this chapter, we'll get our modules to start and stop according to routes. Before we get into the thick of things, let's take a moment to discuss *why* we'd want to start/stop modules according to routes. This is not an approach you should apply to all projects indiscriminately, as in many cases it will introduce extra complexity overhead with little benefit. So when would your app benefit from start/stop functionality? There are 2 main types of cases:

- when your sub-application requires a library that isn't used anywhere else (e.g. a graphing library);
- when your app needs to load a lot of data on initialization, and needs to free the memory when another sub-application loads (e.g. a dashboard with drill-down functionality).

In these cases, we'll be able to load the libraries and/or data when the sub-app starts, and free the memory as the user navigates away from the sub-application (e.g. he's no longer viewing the dashboard requiring advanced visualization and lots of data).

Implementation Strategy

In our current implementation, our modules start automatically with their parent module, and this happens all the way to the main application. In other words, all of the modules we've defined start as soon as the main application is running. We need this to be modified slightly to add module start/stop: obviously, we don't want the sub-apps to start automatically, but only when we tell them to. But implementing this behavior isn't as simple as telling the modules not to start automatically: we need the *routers* for each sub-app to start automatically with the main app, so that they can react to routing events (and start the appropriate module, if necessary).

What we need to solve this challenge is to have a common "router" module that will contain the routing code for all of our sub-apps (and will start automatically); and then we can set `startWithParent` to `false` in the sub-app's main module (e.g. `ContactsApp`). Here's what that looks like:

106

assets/js/apps/contact/contacts_app.js

```
ContactManager.module("ContactsApp", function(ContactsApp, ContactManager,
                                               Backbone, Marionette, $, _){
  ContactsApp.startWithParent = false;

  ContactsApp.onStart = function(){
    console.log("starting contacts app");
  };

  ContactsApp.onStop = function(){
    console.log("stopping contacts app");
  };
});

ContactManager.module("Routers.ContactsApp", function(ContactsAppRouter,
                                    ContactManager, Backbone, Marionette, $, _){
  ContactsAppRouter.Router = Marionette.AppRouter.extend({
    appRoutes: {
      // edited for brevity
    }
  });

  var parseParams = function(params){
    // edited for brevity
  };

  var serializeParams = function(options){
    // edited for brevity
  };

  var API = {
    // edited for brevity
  };

  // navigation event listeners (edited for brevity)

  ContactManager.addInitializer(function(){
    new ContactsAppRouter.Router({
      controller: API
    });
  });
});
```

Let's take a look at our changes:

- on lines 1-12, we define a new Marionette module called "ContactsApp". Within it, we
  - declare (on line 3) that it shouldn't start with its parent app (i.e. the main application);
  - add an `onStart` function (lines 5-7) which will be executed each time the application starts. This is where you would load extra libraries or data required by the sub-app;
  - add an `onStop` function (lines 9-11) to free memory by dereferencing variables that are no longer of use when the user navigates away from the sub-app.
- on line 14, we've renamed our module from `ContactsApp` to `Routers.ContactsApp`. This is because we're going to use the `Routers` Marionette module to hold all of our routing controllers, and it will start (automatically) with the main application;
- the rest of the code hasn't been modified (except for renaming `ContactsApp` to `ContactsAppRouter`, such as on line 37).

What about the sub-modules such a `List`? We're leaving those sub-modules as they are: we need their `startWithParent` property to be `true` (which is the default value). This way, we prevent our top-level `ContactsApp` from starting automatically when `ContactManager` is started, but when we manually start `ContactsApp` its sub-modules (`List`, etc.) will also start (because the are configured to `startWithParent`).

With this code change, we've definind the top-level `ContactsApp` sub-application module within *assets/js/apps/contact/contacts_app.js* to have `startWithParent` set to `false`: it will no longer start, unless we explicitly start it ourselves.

But we're still missing a major piece of the puzzle: we need code to start the sub-app we need, and stop all other sub-apps. Let's imagine we have 2 very large Marionette sub-applications that expose dashboards. One of them deals with customer data (where they live, how often they shop, their profile, etc.), and the other displays aggregate financial information (e.g. sales over the last 2 weeks by customer segment, number of visits, stock turnover, etc.). Both of these sub-applications provide advanced analytics and multiple ways to display and regroup data.

To enhance performance, the most frequently analyzed data is loaded when the sub-app is initialized. This way, when a user wants to change the analysis he's looking at (e.g.from "overall sales" to "single male sales"), the sub-app already has the data available and can display the new analysis rapidly. Unfotrunately, all this data takes up memory, and when the user moves on to a different sub-app, this data isn't necessary anymore: the memory it's using should be freed. This can be accomplished by stopping the current module, and dereferencing the loaded data within the Marionette module's `onStop` method.

With all this module starting and stopping happening, it becomes necessary to have a function we can call to ensure the sub-app wewant to use is active, and stop all the others. Here's what it looks like:

assets/js/app.js

```
ContactManager.getCurrentRoute = function(){
  // edited for brevity
};

ContactManager.startSubApp = function(appName, args){
  var currentApp = ContactManager.module(appName);
  if (ContactManager.currentApp === currentApp){ return; }

  if (ContactManager.currentApp){
   ContactManager.currentApp.stop();
  }

  ContactManager.currentApp = currentApp;
  currentApp.start(args);
};
```

And now, of course, we need to execute this method before any controller action to ensure the sub-app we need is running before we execute one of its methods. In addition, to reduce code duplication, we'll create an `executeAction` function to help us:

assets/js/apps/contact/contacts_app.js

```
ContactsAppRouter.Router = Marionette.AppRouter.extend({
  appRoutes: {
    // edited for brevity
  }
});

var parseParams = function(params){
  // edited for brevity
};

var serializeParams = function(options){
  // edited for brevity
};

var executeAction = function(action, arg){
  ContactManager.startSubApp("ContactsApp");
  action(arg);
  ContactManager.execute("set:active:header", "contacts");
};
```

```
20
21 var API = {
22   listContacts: function(options){
23     executeAction(ContactManager.ContactsApp.List.Controller.listContacts,
24                                                                   options);
25   },
26
27   showContact: function(id){
28     executeAction(ContactManager.ContactsApp.Show.Controller.showContact, id);
29   },
30
31   editContact: function(id){
32     executeAction(ContactManager.ContactsApp.Edit.Controller.editContact, id);
33   }
34 };
```

Modify the "AboutApp" to start its own module, but only when explicitly ordered to. You'll find the solution below.

Modifying the AboutApp

Here's our modified file for the "about" sub-app:

assets/js/apps/about/about_app.js

```
1  ContactManager.module("AboutApp", function(AboutApp, ContactManager,
2                                          Backbone, Marionette, $, _){
3    AboutApp.startWithParent = false;
4
5    AboutApp.onStart = function(){
6      console.log("starting about app");
7    };
8
9    AboutApp.onStop = function(){
10     console.log("stopping about app");
11   };
12 });
13
14 ContactManager.module("Routers.AboutApp", function(AboutAppRouter,
15                         ContactManager, Backbone, Marionette, $, _){
```

```
  AboutAppRouter.Router = Marionette.AppRouter.extend({
    appRoutes: {
      "about" : "showAbout"
    }
  });

  var API = {
    showAbout: function(){
      ContactManager.startSubApp("AboutApp");
      ContactManager.AboutApp.Show.Controller.showAbout();
      ContactManager.execute("set:active:header", "about");
    }
  };

  this.listenTo(ContactManager, "about:show", function(){
    ContactManager.navigate("about");
    API.showAbout();
  });

  ContactManager.addInitializer(function(){
    new AboutAppRouter.Router({
      controller: API
    });
  });
});
```

With these changes in place, you can see the various messages being printed to the console as you navigate around (e.g. by clicking on the links in the header menu). In addition, if you manually enter URLs in the address bar, the modules will also be started/stopped properly (and display the console messages). Success!

Stopping Controllers

Now that we've got our various sub-apps starting and stopping correctly, we still need our controllers to shut down when their containing modules stop. All we need to do is add listeners:

assets/js/apps/about/show/show_controller.js

```
ContactManager.module("AboutApp.Show", function(Show, ContactManager,
                                                 Backbone, Marionette, $, _){
  var Controller = Marionette.Controller.extend({
    // edited for brevity
  });

  Show.Controller = new Controller();

  ContactManager.AboutApp.on("stop", function(){
    Show.Controller.destroy();
  });
});
```

assets/js/apps/contacts/edit/edit_controller.js

```
ContactManager.module("ContactsApp.Edit", function(Edit, ContactManager,
                                                   Backbone, Marionette, $, _){
  var Controller = Marionette.Controller.extend({
    // edited for brevity
  });

  Edit.Controller = new Controller();

  ContactManager.ContactsApp.on("stop", function(){
    Edit.Controller.destroy();
  });
});
```

assets/js/apps/contacts/show/show_controller.js

```
ContactManager.module("ContactsApp.Show", function(Show, ContactManager,
                                                   Backbone, Marionette, $, _){
  var Controller = Marionette.Controller.extend({
    // edited for brevity
  });

  Show.Controller = new Controller();

  ContactManager.ContactsApp.on("stop", function(){
```

```
10      Show.Controller.destroy();
11    });
12  });
```

We are *not* using the `listenTo` method, e.g.

```
1   Show.Controller.listenTo(ContactManager.AboutApp, "stop", function(){
2     Show.Controller.destroy();
3   });
```

The `destroy` call on line 2 will remove all listeners on the controller object, including the one defined on line 1 to listen to the "stop" event. By defining the listener on the `AboutApp` instead (as in the correct code shown in the main text above), the listener will remain even after a controller's `destroy` method is called, because the listener is *not* defined on the controller itself.

Let's add an `onDestroy` method in our last controller, so we can see in the console that the controller has stopped:

assets/js/apps/contacts/list/list_controller.js

```
1  ContactManager.module("ContactsApp.List", function(List, ContactManager,
2                                                      Backbone, Marionette, $, _){
3    var Controller = Marionette.Controller.extend({
4      onDestroy: function(){
5        console.log("destroying controller");
6      },
7      listContacts: function(options){
8        // edited for brevity
9      }
10   });
11
12   List.Controller = new Controller();
13
14   ContactManager.ContactsApp.on("stop", function(){
15     List.Controller.destroy();
16   });
17 });
```

With this code in place, try the following: once the app has started, navigate to *#contacts* to see the paginated list of contacts (make sure you have more than one page of results displayed!). Then, execute this command from the console (and look at the console for the stopping message):

```
ContactManager.ContactsApp.List.Controller.destroy();
```

If you now click on a page number, the corresponding page will be loaded and displayed, but the URL won't be updated. This is because the controller has been stopped and therefore no longer responds to the "page:change" event (which is used to update the "page" parameter in the URL).

Git commit to start/stop modules and controllers:

b9755cfd25e77d98f84a47aaabd78ab7c64ac0ca[107]

[107] https://github.com/davidsulc/marionette-serious-progression-b9755cfd25e77d98f84a47aaabd78ab7c64ac0ca

Working with Model Relationships

In many cases, your application's models with have relationships between one another. In these cases, you want to be able to, for example, load related models as a collection in one of the main model's attributes, and be able to respond to events on that sub-collection. In this chapter, we'll see first how we can manage a sub-collection of related models by hand, and then we'll see how we can leverage Backbone.associations[108] to handle the details for us.

We'll be implementing a so-called self-referential relationship: contacts will be linked to contacts. Contacts will have two new attributes: `acquaintances` (people they know) and `strangers` (people they don't know, i.e. everyone else).

API Properties

We'll continue using the "contacts_paginated" API endpoint for the contacts collection (see properties here), and we'll use a "contacts_paginated/1.json?include_acquaintances=1" API endpoint for our contacts model. Here's an example response:

```
{
  "id":1,
  "firstName":"Abdul",
  "lastName":"Dicki",
  "phoneNumber":"1-600-307-5332",
  "createdAt":"2014-02-01T12:13:23.592Z",
  "updatedAt":"2014-03-24T06:16:35.383Z",
  "acquaintances": [
    {
      "id":5,
      "firstName":"Ursula",
      "lastName":"Gislason",
      "phoneNumber":"1-268-696-3364"
    },
    {
      "id":19,
      "firstName":"Guy",
      "lastName":"Gorczany",
      "phoneNumber":"1-854-233-0676 x27604"
```

[108] https://github.com/dhruvaray/backbone-associations

```
    },
    {
      "id":64,
      "firstName":"Helene",
      "lastName":"Daugherty",
      "phoneNumber":"216-388-8231 x02495"
    },
    {
      "id":75,
      "firstName":"Tyler",
      "lastName":"Kassulke",
      "phoneNumber":"806.213.9177"
    },
    {
      "id":202,
      "firstName":"Courtney",
      "lastName":"O'Conner",
      "phoneNumber":"657.421.7928"
    },
    {
      "id":215,
      "firstName":"Adolfo",
      "lastName":"Gottlieb",
      "phoneNumber":"701.453.8870 x3040"
    }
  ]
}
```


As you can see, when including the "acquaintances=1" query string, our API returns the list of acquaintances along with the main model's data. This list of acquaintances is provided as an array.

Parsing Sub-Models Manually

First, let's change our contact model's `url` property:

assets/js/entities/contact.js

```
Entities.Contact = Entities.BaseModel.extend({
  urlRoot: "contacts",
  url: function(){
    return "contacts_paginated/" + this.get("id") +
           ".json?include_acquaintances=1";
  },

  // edited for brevity
```

As you may remember, this is how URLs work in Backbone models:

- if the model is part of a collection, by default the collection's `url` value will be used to generate the API endpoint by adding the model's id to it;
- if you're using the model outside of a collection, you can specify a `urlRoot` attribute to generate URLs based on the model id (once again by simply adding the model's id: [urlRoot]/[model.id]);
- you can also specify a `url` value or function to override the collection's `url` when it shouldn't be taken into account.

In our case above, we're defining a `url` function that will return the full URL value, as computed by us. As we'll be using the new API endpoint and including the proper query string, we'll obtain a list of acquaintances when fetching a contact's data.

Let's pretend we already have a list of acquaintances defined on our model, and adapt our template to show them.

index.html

```
<script type="text/template" id="contact-view">
  <h1>{{- fullName }}</h1>
  <a href="#contacts/{{- id }}/edit" class="btn btn-small js-edit">
    <i class="icon-pencil"></i>
    Edit this contact
  </a>
  <p><strong>Phone number:</strong> {{- phoneNumber }}</p>

  <h2>Acquaintances</h2>
  {{ if(acquaintances.models.length > 0){ }}
    <ul>
```

```
12      {{ _.each(acquaintances.models, function(a){ }}
13        <li>{{- a.attributes.firstName }} {{- a.attributes.lastName }}</li>
14      {{ }); }}
15      </ul>
16    {{ }else{ }}
17      <p>This contact doesn't know anybody...</p>
18    {{ } }}
19  </script>
```

The example above demonstrates how you can display a collection of elements in a single item view. This can be useful to (e.g.) display dropdowns from collections, or to display non-interactive display elements.

All that's left for us to do is to make sure the incoming data gets put into a collection of acquaintances. Speaking of which, let's define that acquaintance collection:

assets/js/entities/contact.js

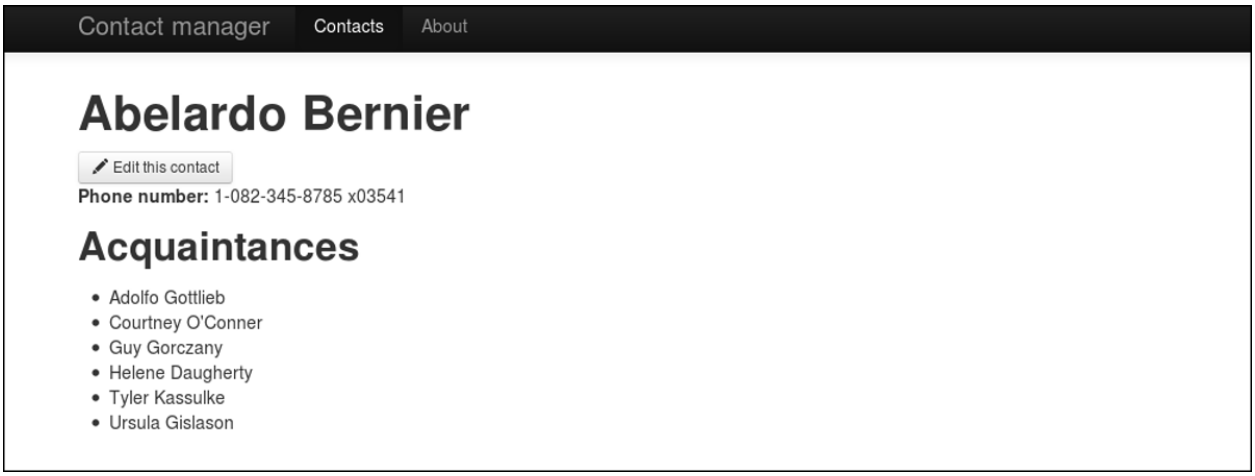
```
1  // edited for brevity
2
3  Entities.AcquaintanceCollection = Backbone.Collection.extend({
4    model: Entities.Contact,
5    comparator: "firstName"
6  });
7
8  Entities.ContactCollection = Backbone.Paginator.requestPager.extend({
9
10   // edited for brevity
```

And now, all that's left for us to do, is parsing the data received from the API, and instantiating a new collection with the acquaintances (lines 9-13):

assets/js/entities/contact.js

```
parse: function(response){
  var data = response;
  if(response && response.contact){
    data = response.contact;
  }
  data.fullName = data.firstName + " ";
  data.fullName += data.lastName;

  if(response.acquaintances){
    data.acquaintances = new Entities.AcquaintanceCollection(
                           response.acquaintances
                         );
  }

  return data;
},
```

With all of this in place, we now see a list of acquaintances when we navigate to a contact's display page:

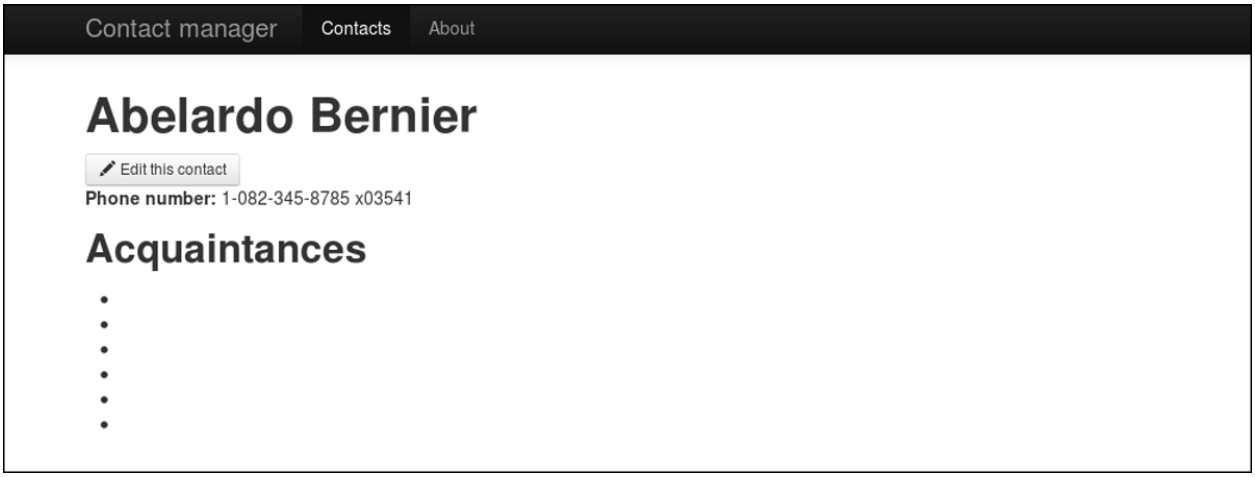


But wait: since we're displaying the full name for each acquaintance, why not use our `fullName` attribute? Let's change our template to do just that:

index.html

```
<script type="text/template" id="contact-view">
  <!-- edited for brevity -->

  <h2>Acquaintances</h2>
  {{ if(acquaintances.models.length > 0){ }}
    <ul>
    {{ _.each(acquaintances.models, function(a){ }}
      <li>{{- a.attributes.fullName }}</li>
    {{ }); }}
    </ul>
  {{ }else{ }}
    <p>This contact doesn't know anybody...</p>
  {{ } }}
</script>
```

Here's what our page looks like after the change:



It turns out that simply adding the collection to the `acquaintances` attribute isn't enough: the data is simply passed to the collection's initializer, but never gets parsed. Since the `fullName` attribute gets computed in the `parse` method, it ends up undefined for each model in the acquaintances collection. We can fix this minor issue by passing an `options` object indicating our collection data should be parsed:

assets/js/entities/contact.js

```
parse: function(response){
  // edited for brevity

  if(response.acquaintances){
    data.acquaintances = new Entities.AcquaintanceCollection(
                           response.acquaintances,
                           { parse: true }
                         );
  }
```

Git commit parsing the contact to load acquaintances, and displaying them:
0d264b3f4a84c5f77ae538b3cfbbc3b461847c78[109]

Paginating Acquaintances

Let's make our "acquaintances" collection a paginated collection: we'll be able to display pages of acquaintances for our popular contacts. We'll be using an API endpoint returning only the contact information (i.e. no embedded acquaintances), and use another API endpoint to return the acquaintances related to a contact.

API Properties

We'll be using the "contacts_paginated" endpoint to fetch our contacts. The return value will resemble (assuming "contacts_paginated/1" was queried):

```
{
  "id":1,
  "firstName":"Abelardo",
  "lastName":"Bernier",
  "phoneNumber":"1-082-345-8785 x03541",
  "createdAt":"2014-02-01T12:13:24.298Z",
  "updatedAt":"2014-03-24T06:16:43.035Z"
}
```

If we now fetch this contact's acquaintances from "contacts_paginated/1/acquaintances", we'd obtain:

[109] https://github.com/davidsulc/marionette-serious-progression-0d264b3f4a84c5f77ae538b3cfbbc3b461847c78

```
{
  "resultCount":6,
  "results": [
    {
      "id":15,
      "firstName":"Adolfo",
      "lastName":"Gottlieb",
      "phoneNumber":"701.453.8870 x3040",
      "url":"http://localhost:3000/contacts/2153.json"
    },
    {
      "id":20,
      "firstName":"Courtney",
      "lastName":"O'Conner",
      "phoneNumber":"657.421.7928",
      "url":"http://localhost:3000/contacts/2022.json"
    },
    {
      "id":95,
      "firstName":"Guy",
      "lastName":"Gorczany",
      "phoneNumber":"1-854-233-0676 x27604",
      "url":"http://localhost:3000/contacts/1959.json"
    },
    {
      "id":164,
      "firstName":"Helene",
      "lastName":"Daugherty",
      "phoneNumber":"216-388-8231 x02495",
      "url":"http://localhost:3000/contacts/1764.json"
    },
    {
      "id":175,
      "firstName":"Tyler",
      "lastName":"Kassulke",
      "phoneNumber":"806.213.9177",
      "url":"http://localhost:3000/contacts/2175.json"
    },
    {
      "id":204,
      "firstName":"Ursula",
      "lastName":"Gislason",
```

```
      "phoneNumber":"1-268-696-3364",
      "url":"http://localhost:3000/contacts/2054.json"
    }
  ]
}
```

Adapting the Code

We'll first update our contact entity to use a paginated collection to contain acquaintances. Since we'll be using the existing contact collection, there's no need for us to define a new collection:

assets/js/entities/contact.js

```
Entities.AcquaintanceCollection = Backbone.Collection.extend({
  model: Entities.Contact,
  comparator: "firstName"
});

Entities.ContactCollection = Backbone.Paginator.requestPager.extend({
  // edited for brevity
```

We're also no longer going to load embedded acquaintances in `parse` (as they'll be queried separately):

assets/js/entities/contact.js

```
_.extend(Entities.Contact.prototype, Backbone.Validation.mixin, {
  parse: function(response){
    var data = response;
    if(response && response.contact){
      data = response.contact;
    }
    data.fullName = data.firstName + " ";
    data.fullName += data.lastName;

    if(response.acquaintances){
      data.acquaintances =
          new Entities.AcquaintanceCollection(
                        response.acquaintances,
                        { parse: true }
                      );
    }

```

```
18      return data;
19    },
20
21    // edited for brevity
```

Let's now move on to what we *are* going to do: change the API endpoint used to fetch models, and define an `acquaintances` attribute when the model get instantiated (which will be a paginated collection):

assets/js/entities/contact.js

```
1  Entities.Contact = Entities.BaseModel.extend({
2    urlRoot: "contacts_paginated",
3
4    initialize: function(){
5      this.set("acquaintances", new Entities.ContactCollection());
6      this.get("acquaintances").paginator_core.url = this.url() + "/acquaintances?\
7  ";
8
9      this.on("change", function(){
10       this.set("fullName", this.get("firstName") + " " + this.get("lastName"));
11     });
12   },
13
14   // edited for brevity
```

Notice that on line 6, we add "?" to the URL: as we'll be dealing with a paginated collection, adding the question mark will allow us to add query string parameters to configure the data that gets returned to us (page number, filter criterion, etc.).

With our contact entity updated, let's move on to our show controller:

assets/js/apps/contacts/show/show_controller.js

```
1  $.when(fetchingContact).done(function(contact){
2    contact.get("acquaintances").fetch().then(function(){
3      var contactView = new Show.Contact({
4        model: contact
5      });
6
7      Show.Controller.listenTo(contactView, "contact:edit", function(contact){
8        ContactManager.trigger("contact:edit", contact.get("id"));
9      });
10
11     ContactManager.mainRegion.show(contactView);
12   });
13 }).fail(function(response){
```

The difference here is line 2: we fetch the contact's acquaintances and wait for them to load before displaying the page.

The `fetch(...).then(...)` syntax on line 2 above is equivalent (when using deferreds) to

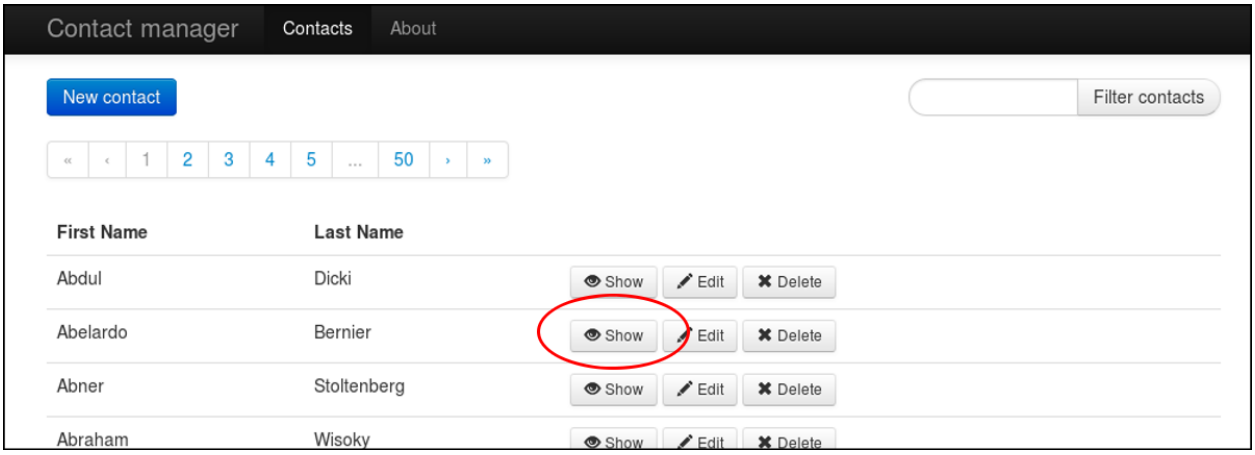
```
var acquaintancesFetched = contact.get("acquaintances").fetch();
$.when(acquaintancesFetched).done(...);
```

which is what we've used so far. The main difference when using `$.when` is that it can receive a non-deferred argument (e.g. a simple string), in which case it will execute the `done` callback immediately. It is usually not possible to call a `then` method on non-deferreds.

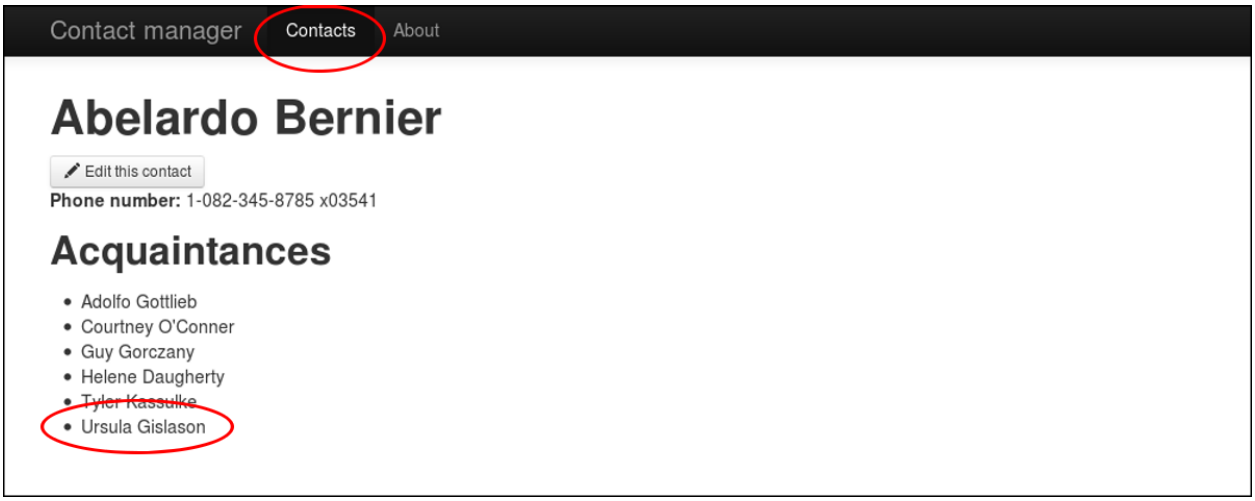
In addition, the `$.when` syntax can be used to wait for multiple deferreds to return before proceeding (i.e. a simple synchronization mechanism). We'll put this to use later.

Fixing the Paginated Collection

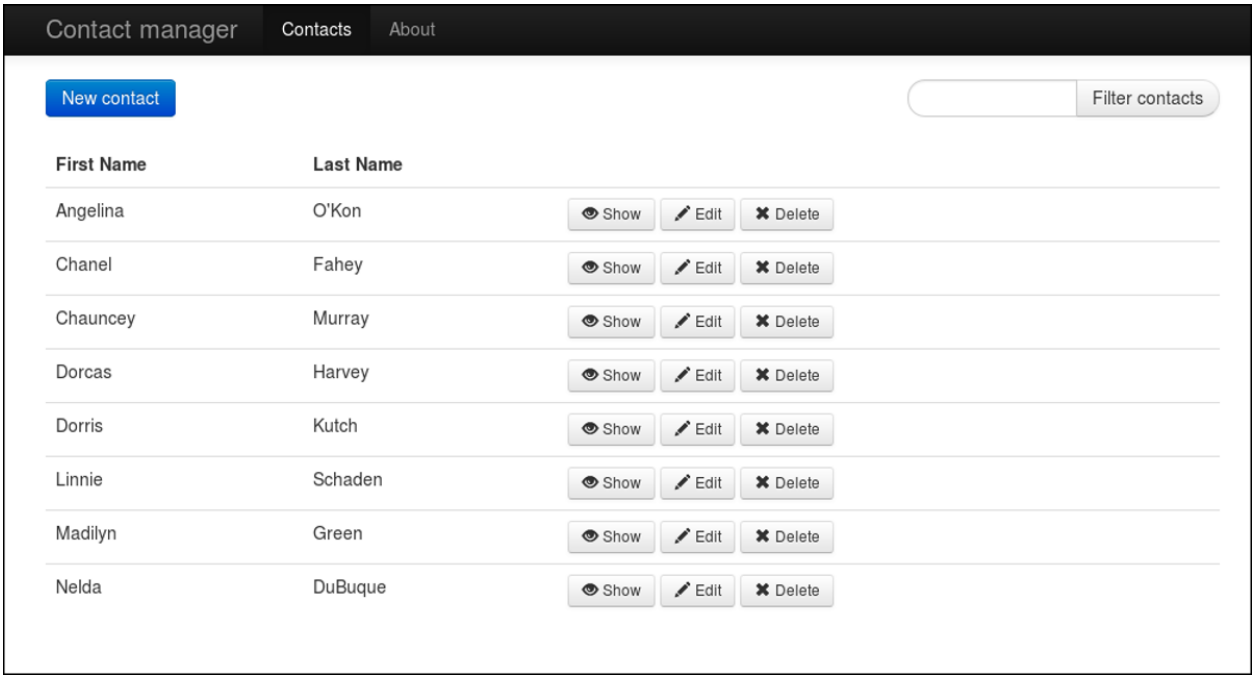
There's a nasty bug in our app relating to collections. Let's walk through it.



Display the list of contacts and click a "show" button

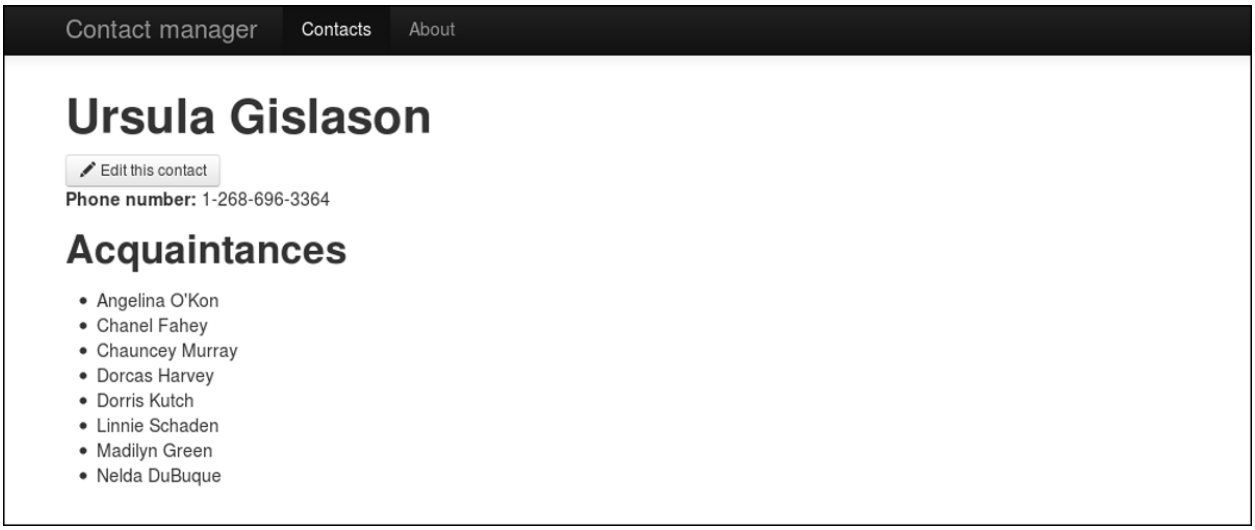


Select the "contacts" menu item to return to the list of contacts (take notice of the last acquaintance)



There's now only one page of contacts, and it doesn't look anything like the page we had in the first screen shot…

What's going on? What contacts are getting displayed in the last screen shot? They actually match the last acquaintance's (i.e. Ursula Gislason, see above) own acquaintances:



Ursula Gislason's acquaintances. Do they look familiar?

So how did this happen? Well, let's look at our contact collection's definition:

assets/js/entities/contact.js

```
Entities.ContactCollection = Backbone.Paginator.requestPager.extend({
  model: Entities.Contact,

  initialize: function(models, options){
    // edited for brevity
  },

  comparator: "firstName",
  paginator_core: {
    dataType: "json",
    url: "contacts_paginated?"
  },
  paginator_ui: {
    firstPage: 1,
    currentPage: 1,
    perPage: 10,
    pagesInRange: 2
  },
  server_api: {
    count: function() { return this.perPage },
    offset: function() {
      return ((this.parameters.get("page") || 1) - 1) * this.perPage
    },
    filter: function() { return this.parameters.get("criterion"); }
  }
});
```

The issue we've got is that all paginator configuration is defined as properties of `ContactCollection`. Therefore, when a contact collection changes (e.g.) `this.paginator_ui.currentPage`, the `paginator_ui` object it has access to is updated. Unfortunately, this object is shared: all contact collections will then have their current state and configuration modified. That's clearly not what we want, so let's instead have each contact collection create its own private pagination configuration in the `initialize` function (so it's tied only to that instance):

assets/js/entities/contact.js

```
Entities.ContactCollection = Backbone.Paginator.requestPager.extend({
  model: Entities.Contact,

  initialize: function(models, options){
    options || (options = {});

    var params = options.parameters || { page: 1 };
    this.parameters = new Backbone.Model(params);

    this.paginator_core = {
      dataType: "json",
      url: "contacts_paginated?"
    };
    this.paginator_ui = {
      firstPage: 1,
      currentPage: 1,
      perPage: 10,
      pagesInRange: 2
    };
    this.server_api = {
      count: function() { return this.perPage },
      offset: function() {
        return ((this.parameters.get("page") || 1) - 1) * this.perPage
      },
      filter: function() { return this.parameters.get("criterion"); }
    };

    var self = this;
    this.listenTo(this.parameters, "change", function(model){
      if(_.has(model.changed, "criterion")){
        self.server_api.filter = self.parameters.get("criterion");
      }
      $.when(this.pager()).done(function(){
        self.trigger("page:change:after");
      });
    });

    this.on("sync", function(){
      this.sort({silent: true});
      this.trigger("reset");
    });
```

```
42    },
43
44    comparator: "firstName"
45  - comparator: "firstName",
46  - paginator_core: {
47  -   dataType: "json",
48  -   url: "contacts_paginated?"
49  - },
50  - paginator_ui: {
51  -   firstPage: 1,
52  -   currentPage: 1,
53  -   perPage: 10,
54  -   pagesInRange: 2
55  - },
56  - server_api: {
57  -   count: function() { return this.perPage },
58  -   offset: function() {
59  -     return ((this.parameters.get("page") || 1) - 1) * this.perPage
60  -   },
61  -   filter: function() { return this.parameters.get("criterion"); }
62  - }
63  });
```

If you try the scenario above again, you'll find that our app now behaves as expected. Whew!

Complex Nested Views

What if one of our contacts is a social butterfly and has a lot of acquaintances? We certainly wouldn't want to have hundreds of displayed contacts taking up room on the page. You know what would be awesome to solve that problem? A paginated view. Aaawwwwwww yiiisssssss. But we won't stop there: we'll also want to display a contact's "strangers" list (i.e. the people he *doesn't* know). And it should also be paginated... To give you an appreciation of how hardcore our "show" view is about to become, let's list the various views we'll have once we're done:

- the main "show" view, which will be a layout, containing
  - an "acquaintances" paginated view, which is itself a layout containing
    * a "pagination controls" item view
    * an arbitrary "content view", which could itself be a collection view containing sub views
  - a "strangers" paginated view, which is itself a layout containing
    * a "pagination controls" item view

* an arbitrary "content view", which could itself be a collection view containing sub views

In other words, we're going to have pretty complex nested views here, but we'll see how it's easily managed with Marionette.

The rule of thumb when working with nested views, as you might guess from above, is to use layouts to group the views in a hierarchy. Then, each view is responsible for displaying its sub-views (and possibly proxying their events).

Speaking of which, let's start by turning our "show" view into a layout, so it can later hold paginated sub-views:

assets/js/apps/contacts/show/show_view.js

```
~~Show.Contact = Marionette.ItemView.extend({~~
Show.Contact = Marionette.LayoutView.extend({
  template: "#contact-view",

  events: {
    "click a.js-edit": "editClicked"
  }
});
```

We can now use a collection view to display our acquaintances (later, we'll display that collection view within a paginated view).

index.html

```
<script type="text/template" id="contact-view">
  <h1>{{- fullName }}</h1>
  <a href="#contacts/{{- id }}/edit" class="btn btn-small js-edit">
    <i class="icon-pencil"></i>
    Edit this contact
  </a>
  <p><strong>Phone number:</strong> {{- phoneNumber }}</p>

  <h2>Acquaintances</h2>
  <div id="acquaintances-region"></div>
~~  {{ if(acquaintances.models.length > 0){ }}~~
~~    <ul>~~
~~    {{ _.each(acquaintances.models, function(a){ }}~~
```

```
      <li>{{- a.attributes.fullName }}</li>
    {{ }); }}
    </ul>
  {{ }else{ }}
    <p>This contact doesn't know anybody...</p>
  {{ } }}
</script>

<script type="text/template" id="contact-acquaintance-view">
  {{- fullName }}
</script>
```

assets/js/apps/contacts/show/show_view.js

```
Show.Contact = Marionette.LayoutView.extend({
  template: "#contact-view",

  regions: {
    acquaintancesRegion: "#acquaintances-region"
  },

  events: {
    "click a.js-edit": "editClicked"
  }
});

_.extend(Show.Contact.prototype, {
  // edited for brevity
});

Show.Acquaintance = Marionette.ItemView.extend({
  tagName: "li",
  template: "#contact-acquaintance-view"
});

Show.Acquaintances = Marionette.CollectionView.extend({
  tagName: "ul",
  childView: Show.Acquaintance
});
```

assets/js/apps/contacts/show/show_controller.js

```
$.when(fetchingContact).done(function(contact){
  contact.get("acquaintances").fetch().then(function(){
    var contactView = new Show.Contact({
      model: contact
    });

    Show.Controller.listenTo(contactView, "show", function(){
      var acquaintancesView = new Show.Acquaintances({
        collection: contact.get("acquaintances")
      });
      contactView.acquaintancesRegion.show(acquaintancesView);
    });

    // edited for brevity
```

For simplicity, our collection view doesn't define an "empty view" (see docs[110]) but feel free to define one. An empty view was defined for the contact "list" view (*assets/js/apps/-contacts/list/list_view.js*), if you're looking for inspiration.

Adding Strangers to the Mix

Let's display a contact's "strangers" (i.e. people he doesn't know) along with the acquaintances. First, we need a collection for them:

assets/js/entities/contact.js

```
Entities.Contact = Entities.BaseModel.extend({
  urlRoot: "contacts_paginated",

  initialize: function(){
    this.set("acquaintances", new Entities.ContactCollection());
    this.get("acquaintances").paginator_core.url = this.url() + "/acquaintances?\
";

    this.set("strangers", new Entities.ContactCollection());
    this.get("strangers").paginator_core.url = this.url() + "/strangers?";
```

[110] https://github.com/marionettejs/backbone.marionette/blob/master/docs/marionette.collectionview.md#collectionviews-emptyview

```
11
12      this.on("change", function(){
13        this.set("fullName", this.get("firstName") + " " + this.get("lastName"));
14      });
15    },
16
17    // edited for brevity
```

We'll need a region to display our strangers:

index.html

```
1  <script type="text/template" id="contact-view">
2    <h1>{{- fullName }}</h1>
3    <a href="#contacts/{{- id }}/edit" class="btn btn-small js-edit">
4      <i class="icon-pencil"></i>
5      Edit this contact
6    </a>
7    <p><strong>Phone number:</strong> {{- phoneNumber }}</p>
8
9    <h2>Acquaintances</h2>
10   <div id="acquaintances-region"></div>
11
12   <h2>Strangers</h2>
13   <div id="strangers-region"></div>
14 </script>
```

assets/js/apps/contacts/show/show_view.js

```
1  Show.Contact = Marionette.LayoutView.extend({
2    template: "#contact-view",
3
4    regions: {
5      acquaintancesRegion: "#acquaintances-region",
6      strangersRegion: "#strangers-region"
7    },
8
9    events: {
10     "click a.js-edit": "editClicked"
11   }
12 });
```

We still need to display these "stranger" contacts in some sort of view. For now, we'll reuse the `Acquaintances` view (of course, if we were going to use it permanently, it should be renamed). But we still have one challenge left: we'd like to wait until both the acquaintances *and* the strangers collections are loaded before displaying the "show" view. Luckily, using deferreds we can accomplish that in a readable manner, without any nested callbacks:

assets/js/apps/contacts/show/show_controller.js

```
1  $.when(fetchingContact).done(function(contact){
2    contact.get("acquaintances").fetch().then(function(){
3    var acquaintancesFetched = contact.get("acquaintances").fetch(),
4        strangersFetched = contact.get("strangers").fetch();
5    $.when(acquaintancesFetched, strangersFetched).done(function(){
6      var contactView = new Show.Contact({
7        model: contact
8      });
9
10     Show.Controller.listenTo(contactView, "show", function(){
11       var acquaintancesView = new Show.Acquaintances({
12         collection: contact.get("acquaintances")
13       });
14       var strangersView = new Show.Acquaintances({
15         collection: contact.get("strangers")
16       });
17       contactView.acquaintancesRegion.show(acquaintancesView);
18       contactView.strangersRegion.show(strangersView);
19     });
20
21     // edited for brevity
```

On lines 3-5, you can see that the `$.when(...)` syntax allows us to provide several promises and will execute the success callback only once *all* promises have returned successfully. Nice, isn't it?

Paginating the Sub-Views

Let's start with a little refactoring: we'll add the "acquaintances" and "strangers" collections to the contact model when we request it:

assets/js/entities/contact.js

```
getContactEntity: function(contactId, options){
  // edited for brevity
  var response = contact.fetch(_.omit(options, 'success', 'error'));
  response.done(function(){
    contact.set("acquaintances", new Entities.ContactCollection());
    contact.get("acquaintances").paginator_core.url =
                                              contact.url() + "/acquaintances?";

    contact.set("strangers", new Entities.ContactCollection());
    contact.get("strangers").paginator_core.url = contact.url() + "/strangers?";

    defer.resolveWith(response, [contact]);
  });
  response.fail(function(){
    defer.rejectWith(response, arguments);
  });
  return defer.promise();
}
```

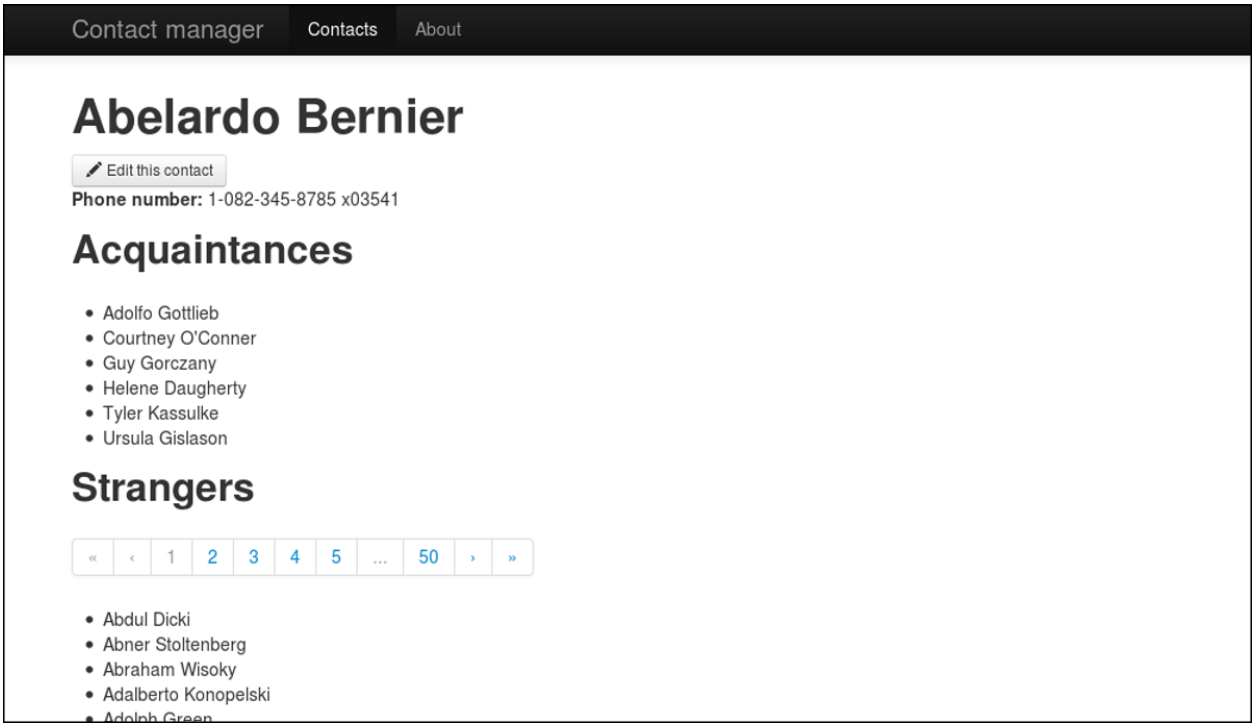
And now for the paginated views:

assets/js/apps/contacts/show/show_controller.js

```
Show.Controller.listenTo(contactView, "show", function(){
  var acquaintancesView = new Show.Acquaintances({
    collection: contact.get("acquaintances")
  });
  var strangersView = new Show.Acquaintances({
    collection: contact.get("strangers")
  });
  var acquaintances = contact.get("acquaintances");
  var acquaintancesView = new ContactManager.Common.Views.PaginatedView({
    collection: acquaintances,
    mainView: Show.Acquaintances
  });

  var strangers = contact.get("strangers");
  var strangersView = new ContactManager.Common.Views.PaginatedView({
    collection: strangers,
    mainView: Show.Acquaintances
  });
```

```
19    contactView.acquaintancesRegion.show(acquaintancesView);
20    contactView.strangersRegion.show(strangersView);
21  });
```

Now, our sub-views have some pagination sexyness:



Paginated sub-views

Don't be alarmed if acquaintances don't have pagination: it's likely that the contact you're looking at doesn't have enough contacts as "acquaintances" to trigger the display of pagination controls.

Optional Pagination URLs

If you hover over our sub-views' pagination controls, you'll notice that they have a strange, nonsensical URL. In this case, we don't want to have specific URLs depending on which page a sub-view is displaying. Let's fix that by giving our pagination controls a URL only if a `urlBase` option is provided to the paginated view:

assets/js/common/views.js

```
serializeData: function(){
  var data = this.paginatedCollection.info(),
      url = this.urlBase,
      criterion = this.paginatedCollection.parameters.get("criterion");
  if(criterion){
    url += "criterion:" + criterion + "+";
  }
  url += "page:";
  if(url){
    if(criterion){
      url += "criterion:" + criterion + "+";
    }
    url += "page:";
  }
  data.urlBase = url;

  return data;
}
```

We also need to update our template:

index.html

```
<script type="text/template" id="pagination-controls">
  {{ if(totalPages > 1){ }}
    <ul>
      {{ if(currentPage > 1){ }}
        <li><a href="#{{- urlBase + 1 }}" class="navigatable"
                                              data-page="1">&laquo;</a></li>
        <li><a href="#{{- urlBase + previous }}" class="navigatable"
                                    data-page="{{- previous }}">&lsaquo;</a></li>
        <li><a href="#{{- urlBase ? urlBase + 1 : "" }}" class="navigatable"
                                                 data-page="1">&laquo;</a></li>
        <li><a href="#{{- urlBase ? urlBase + previous  : "" }}"
              class="navigatable" data-page="{{- previous }}">&lsaquo;</a></li>
      {{ }else{ }}
        <li class="disabled"><a href="#">&laquo;</a></li>
        <li class="disabled"><a href="#">&lsaquo;</a></li>

        <!-- edited for brevity -->

```

```
        {{ if(page === currentPage){ }}
          <li class="active disabled"><a href="#">{{- page }}</a></li>
        {{ }else{ }}
          <li><a href="#{{- urlBase + page }}" class="navigatable"
                                  data-page="{{- page }}">{{- page }}</a></li>
          <li><a href="#{{- urlBase ? urlBase + page  : "" }}"
              class="navigatable" data-page="{{- page }}">{{- page }}</a></li>
        {{ } }}
      {{ }); }}

      {{ if(pageSet[pageSet.length - 1] !== lastPage){ }}
        <li class="disabled"><a href="#">...</a></li>
        <li><a href="#{{- urlBase + lastPage }}" class="navigatable"
                            data-page="{{- lastPage }}">{{- lastPage }}</a></li>
        <li><a href="#{{- urlBase ? urlBase + lastPage  : "" }}"
               class="navigatable" data-page="{{- lastPage }}">
                                                    {{- lastPage }}</a></li>
      {{ } }}

      {{ if(currentPage !== lastPage){ }}
        <li><a href="#{{- urlBase + next }}" class="navigatable"
                                data-page="{{- next }}">&rsaquo;</a></li>
        <li><a href="#{{- urlBase + lastPage }}" class="navigatable"
                              data-page="{{- lastPage }}">&raquo;</a></li>
        <li><a href="#{{- urlBase ? urlBase + next  : "" }}"
             class="navigatable" data-page="{{- next }}">&rsaquo;</a></li>
        <li><a href="#{{- urlBase ? urlBase + lastPage  : "" }}"
             class="navigatable" data-page="{{- lastPage }}">&raquo;</a></li>
      {{ }else{ }}
        <li class="disabled"><a href="#">&rsaquo;</a></li>
        <li class="disabled"><a href="#">&raquo;</a></li>
```

Great, now our sub-views will still paginate, but they won't have meaningless URLs.

Adding and Removing Acquaintances

Now that we've got our acquaintances and strangers displaying as we want them, let's make our users able to add or remove acquaintances. To achieve this, our API provides a contacts/1/acquaintances/9 endpoint. By POSTing to that URL, we'll add contact with id 9 to the acquaintances for the contact with id 1. Similarly, by sending a DELETE request to that URL, we'd remove contact 9 from contact 1's acquaintances (thereby adding contact 9 to the "strangers" list).

The first thing we need to do, is to add some "remove" and "add" buttons to our "acquaintances" and "strangers" views, respectiveley. And since our views will now behave differently, strangers will get their own view.

Adding buttons and templates (index.html)

```
<script type="text/template" id="contact-acquaintance-view">
  {{- fullName }} <button class="js-remove-acquaintance">Remove</button>
</script>

<script type="text/template" id="contact-stranger-view">
  {{- fullName }} <button class="js-add-acquaintance">Acquaintance</button>
</script>
```

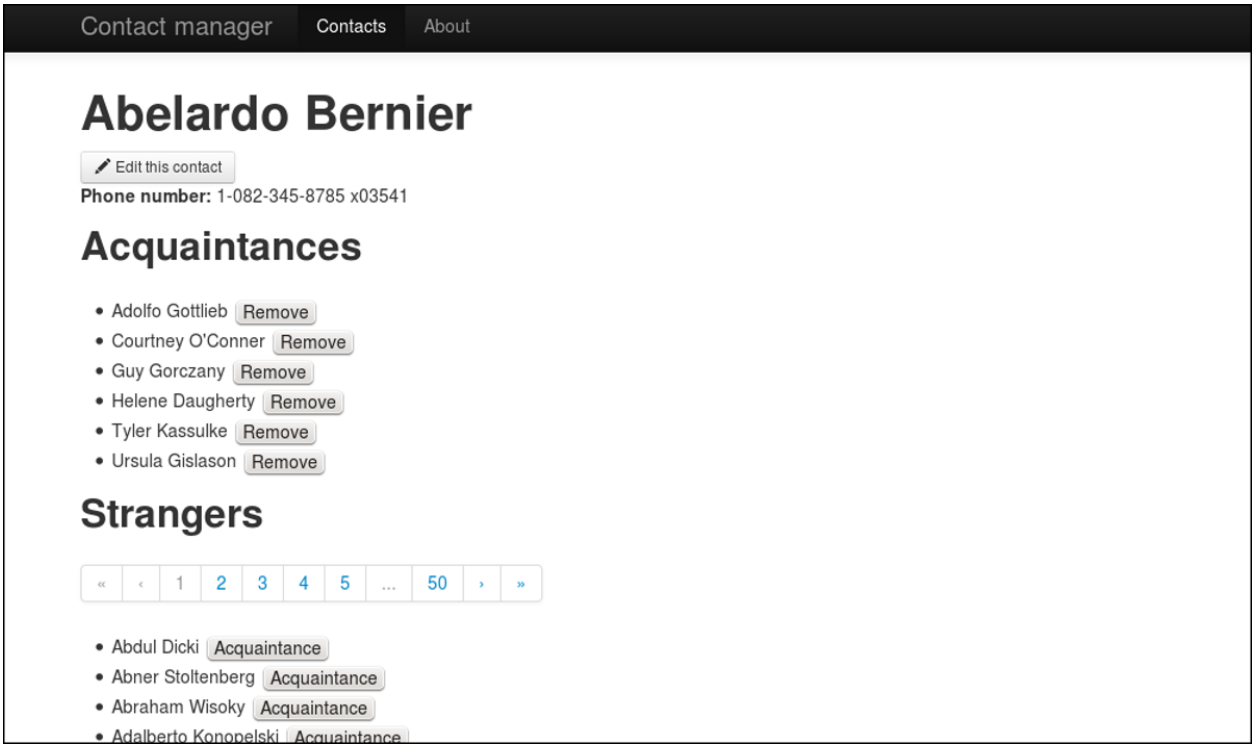
Acquaintance and stranger views (assets/js/apps/contacts/show/show_view.js)

```
Show.Acquaintance = Marionette.ItemView.extend({
  tagName: "li",
  template: "#contact-acquaintance-view",
  triggers: {
    "click .js-remove-acquaintance": "acquaintance:remove"
  }
});

Show.Acquaintances = Marionette.CollectionView.extend({
  tagName: "ul",
  childView: Show.Acquaintance
});

Show.Stranger = Marionette.ItemView.extend({
  tagName: "li",
  template: "#contact-stranger-view",
  triggers: {
    "click .js-add-acquaintance": "acquaintance:add"
  }
});

Show.Strangers = Marionette.CollectionView.extend({
  tagName: "ul",
  childView: Show.Stranger
});
```

Let's get our views displayed on the page:

assets/js/apps/contacts/show/show_controller.js

```
var acquaintancesView = new ContactManager.Common.Views.PaginatedView({
  collection: acquaintances,
  mainView: Show.Acquaintances
  mainView: Show.Acquaintances,
  propagatedEvents: ["childview:acquaintance:remove"]
});

var strangers = contact.get("strangers");
var strangersView = new ContactManager.Common.Views.PaginatedView({
  collection: strangers,
  mainView: Show.Acquaintances
  mainView: Show.Strangers,
  propagatedEvents: ["childview:acquaintance:add"]
});
```



Our sub-views, now with buttons

Our views are now triggering events when the user clicks on buttons to add and remove contacts from the acquaintances list. How should we process these? Well, the first step is to add/remove models from the contact's "acquaintances" collection:

assets/js/apps/contacts/show/show_controller.js

```
var strangersView = new ContactManager.Common.Views.PaginatedView({
  // edited for brevity
});

Show.Controller.listenTo(acquaintancesView, "childview:acquaintance:remove",
                                                        function(view, args){
  contact.get("acquaintances").remove(args.model);
});
Show.Controller.listenTo(strangersView, "childview:acquaintance:add",
                                                        function(view, args){
  contact.get("acquaintances").add(args.model);
});
```

So far, so good. What needs to happen now, is that we fire off the proper ajax requests when a contact gets added or removed from the acquaintances collection:

assets/js/apps/contacts/show/show_controller.js

```
Show.Controller.listenTo(contactView, "show", function(){
  var acquaintances = contact.get("acquaintances");
  var strangers = contact.get("strangers");

  var acquaintancesUrl = _.result(contact, "url") + "/acquaintances/";
  contact.listenTo(acquaintances, "add", function(model){
    $.ajax({
      url: acquaintancesUrl + model.get("id"),
      type: "POST",
      dataType: "json",
      success: function(){
        contact.get("strangers").remove(model);
      }
    });
  });
  contact.listenTo(acquaintances, "remove", function(model){
    $.ajax({
      url: acquaintancesUrl + model.get("id"),
      type: "DELETE",
      dataType: "json",
      success: function(){
        contact.get("strangers").add(model);
      }
```

```
24      });
25    });
26
27    // edited for brevity
```

As you can tell, the process is relatively straightforward:

1. build the main "acquaintances URL", using the contact `url` attribute or function (line 5);
2. respond to the "add" and "remove" collection events by sending a Post or DELETE request to the proper API endpoint (which depends on the model added/removed);
3. update the strangers collection appropriately (the acquaintances collection has already been updated, since it triggered all of these actions).

We're not handling API failures here for simplicity (and also because only internal server errors could cause errors).

Git commit managing a contact's acquaintances:

44428da21f951e110df1deeff6de62ce1a9da43d[111]

Using Backbone.Associations

So far we've handled the entire sub-model challenge on our own, but there are libraries that exist to do some of the work for us. In this section, we'll see how we can use Backbone.associations[112] to help out.

Another library you can look into using is Backbone.relational[113]

First, we need to actually load the library, so grab it from here[114], copy it into *assets/js/vendor/backbone-associations.js*, and include it in *index.html*:

[111] https://github.com/davidsulc/marionette-serious-progression-44428da21f951e110df1deeff6de62ce1a9da43d
[112] https://github.com/dhruvaray/backbone-associations
[113] http://backbonerelational.org/
[114] https://raw.githubusercontent.com/davidsulc/marionette-serious-progression-app/master/assets/js/vendor/backbone-associations.js

index.html

```
<script src="./assets/js/vendor/jquery.js"></script>
<script src="./assets/js/vendor/jquery-ui-1.10.3.js"></script>
<script src="./assets/js/vendor/json2.js"></script>
<script src="./assets/js/vendor/underscore.js"></script>
<script src="./assets/js/vendor/backbone.js"></script>
<script src="./assets/js/vendor/backbone-associations.js"></script>
<script src="./assets/js/vendor/backbone.validation.js"></script>
<!-- edited for brevity -->
```

Let's configure out contact to used associated models:

assets/js/entities/contact.js

```
~~Entities.Contact = Entities.BaseModel.extend({~~
Entities.Contact = Backbone.AssociatedModel.extend({
  urlRoot: "contacts_paginated",

  relations: [
    {
      type: Backbone.Many,
      key: "acquaintances",
      collectionType: "ContactManager.Entities.ContactCollection"
    },
    {
      type: Backbone.Many,
      key: "strangers",
      collectionType: "ContactManager.Entities.ContactCollection"
    }
  ],

  initialize: function(){
    var self = this;
    this.get("acquaintances").paginator_core.url = function(){
      return _.result(self, "url") + "/acquaintances?";
    };
    this.get("strangers").paginator_core.url = function(){
      return _.result(self, "url") + "/strangers?";
    };

    this.on("change", function(){
      this.set("fullName", this.get("firstName") + " " + this.get("lastName"));
```

```
29      });
30    },
31
32    defaults: {
33      firstName: "",
34      lastName: "",
35      phoneNumber: "",
36
37      acquaintances: [],
38      strangers: [],
39      changedOnServer: false
40    }
41  });
```

On line 2, we need to extend from an `AssociatedModel` to have access to the additional association functionality. We then configure 2 "many" relationships for our acquaintances and strangers on lines 5-16 (you can also define one-to-one relationships[115]). We also configure the acquaintances and strangers collections when a model gets initialized (lines 19-25). Finally, we define empty arrays for each of our associations (lines 37-38).

Without the empty arrays as default values on lines 37-38, initial "get" calls on the acquaintances and strangers associations will *not* work.

Since we're using Backbone.associations to handle the sub-models, we no longer need to do it by hand:

assets/js/entites/contact.js

```
1  getContactEntity: function(contactId, options){
2    var contact = new Entities.Contact({id: contactId});
3    var defer = $.Deferred();
4    options || (options = {});
5    defer.then(options.success, options.error);
6    var response = contact.fetch(_.omit(options, 'success', 'error'));
7    response.done(function(){
8  -   contact.set("acquaintances", new Entities.ContactCollection());
9  -   contact.get("acquaintances").paginator_core.url =
10 -                                   contact.url() + "/acquaintances?";
11
12 -   contact.set("strangers", new Entities.ContactCollection());
```

[115] http://dhruvaray.github.io/backbone-associations/specify-associations.html#sa-one

```
13      contact.get("strangers").paginator_core.url =
14                                          contact.url() + "/strangers?";
15      defer.resolveWith(response, [contact]);
16    });
17    response.fail(function(){
18      defer.rejectWith(response, arguments);
19    });
20    return defer.promise();
21  }
```

The thing is, we really only want to have the overhead of creating acquaintances and strangers collections *when we need them*. After all, there's no reason to instantiate a whole bunch of dependent collections (for acquaintances and strangers) when we're listing contacts on the "list" page... So let's configure the associations only if it's explicitly requested in the `options`:

assets/js/entities/contact.js

```
1  Entities.Contact = Backbone.AssociatedModel.extend({
2
3    // edited for brevity
4
5    initialize: function(options){
6      options || (options = {});
7
8      if(options.configureRelationships){
9        var self = this;
10       this.get("acquaintances").paginator_core.url = function(){
11         return _.result(self, "url") + "/acquaintances?";
12       };
13       this.get("strangers").paginator_core.url = function(){
14         return _.result(self, "url") + "/strangers?";
15       };
16     }
17
18     this.on("change", function(){
19       this.set("fullName", this.get("firstName") + " " + this.get("lastName"));
20     });
21   },
22
23   // edited for brevity
```

Don't forget to add the `options` argument to the `initialize` function's signature on line 5.

And then, we need to request the associations to be configured in our `getContactEntity` handler:

assets/js/entities/contact.js

```
getContactEntity: function(contactId, options){
  var contact = new Entities.Contact({id: contactId});
  var contact = new Entities.Contact({
    id: contactId,
    configureRelationships: true
  });
  var defer = $.Deferred();

  // edited for brevity
```

With all of the preliminary work out of the way, let's finish implementing functional acquaintances and strangers. First, when configuring associations, we also need to handle the updates (i.e. adding/removing acquaintances):

assets/js/entities/contact.js

```
Entities.Contact = Backbone.AssociatedModel.extend({
  urlRoot: "contacts_paginated",

  relations: [
    // edited for brevity
  ],

  initialize: function(options){
    options || (options = {});

    if(options.configureRelationships){
      var self = this,
          acquaintances = this.get("acquaintances"),
          acquaintancesUrl = _.result(this, "url") + "/acquaintances";

      acquaintances.paginator_core.url = acquaintancesUrl + "?";
      acquaintancesUrl = acquaintancesUrl + "/";
      this.listenTo(acquaintances, "add", function(model){
        $.ajax({
          url: acquaintancesUrl + model.get("id"),
          type: "POST",
          dataType: "json",
          success: function(){
```

```
24              self.get("strangers").remove(model);
25            },
26            error: function(){
27              acquaintances.remove(model);
28            }
29          });
30        });
31        this.listenTo(acquaintances, "remove", function(model){
32          $.ajax({
33            url: acquaintancesUrl + model.get("id"),
34            type: "DELETE",
35            dataType: "json",
36            success: function(){
37              self.get("strangers").add(model);
38            },
39            error: function(){
40              acquaintances.add(model);
41            }
42          });
43        });
44
45
46        this.get("strangers").paginator_core.url = function(){
47          return _.result(self, "url") + "/strangers?";
48        };
49      }
50
51      // edited for brevity
```

We're only listening to "add" and "remove" events on the associated collections, but you could do a lot more. For example, you could respond to the first acquaintance's last name changing:

```
this.listenTo(acquaintances, "change:acquaintances[0].lastName", doSomething);
```

See the eventing documentation[116] for more info on the possibilities.

Since we're now handling the updates to the acquaintances collection from within the contact, we can remove that code from our controller:

[116] http://dhruvaray.github.io/backbone-associations/events.html

assets/js/apps/contact/show/show_controller.js

```
Show.Controller.listenTo(contactView, "show", function(){
  var acquaintances = contact.get("acquaintances");
  var strangers = contact.get("strangers");

  var acquaintancesUrl = _.result(contact, "url") + "/acquaintances/";
  contact.listenTo(acquaintances, "add", function(model){
    $.ajax({
      url: acquaintancesUrl + model.get("id"),
      type: "POST",
      dataType: "json",
      success: function(){
        contact.get("strangers").remove(model);
      }
    });
  });
  contact.listenTo(acquaintances, "remove", function(model){
    $.ajax({
      url: acquaintancesUrl + model.get("id"),
      type: "DELETE",
      dataType: "json",
      success: function(){
        contact.get("strangers").add(model);
      }
    });
  });

  var acquaintancesView = new ContactManager.Common.Views.PaginatedView({
    // edited for brevity
```

Finally, since we're listening to "add" events on the acquaintances and strangers collections (to send ajax requests to the API, creating new acquaintance relationships) we need to *silently* fetch our collection to avoid triggering these unnecessary requests:

assets/js/apps/contact/show/show_controller.js

```
$.when(fetchingContact).done(function(contact){
  var acquaintancesFetched = contact.get("acquaintances").fetch(),
      strangersFetched = contact.get("strangers").fetch();
  var acquaintancesFetched = contact.get("acquaintances").fetch({silent: true}),
      strangersFetched = contact.get("strangers").fetch({silent: true});

  // edited for brevity
```

Refactoring

Let's simplify and lighten our "show" controller a bit, starting with moving the listener code into an `onShow` function:

assets/js/apps/contacts/show/show_controller.js

```
$.when(fetchingContact).done(function(contact){
  var acquaintancesFetched = contact.get("acquaintances").fetch({silent: true}),
      strangersFetched = contact.get("strangers").fetch({silent: true});
  $.when(acquaintancesFetched, strangersFetched).done(function(){
    var contactView = new Show.Contact({
      model: contact
    });

    Show.Controller.listenTo(contactView, "show", function(){
      var acquaintances = contact.get("acquaintances");
      var strangers = contact.get("strangers");

      var acquaintancesView = new ContactManager.Common.Views.PaginatedView({
        collection: acquaintances,
        mainView: Show.Acquaintances,
        propagatedEvents: ["childview:acquaintance:remove"]
      });

      var strangersView = new ContactManager.Common.Views.PaginatedView({
        collection: strangers,
        mainView: Show.Strangers,
        propagatedEvents: ["childview:acquaintance:add"]
      });

      Show.Controller.listenTo(acquaintancesView,
```

```
                                "childview:acquaintance:remove", function(view, args){
        contact.get("acquaintances").remove(args.model);
    });
    Show.Controller.listenTo(strangersView, "childview:acquaintance:add",
                                                        function(view, args){
        contact.get("acquaintances").add(args.model);
    });

    contactView.acquaintancesRegion.show(acquaintancesView);
    contactView.strangersRegion.show(strangersView);
  });

  Show.Controller.listenTo(contactView, "contact:edit", function(contact){
    ContactManager.trigger("contact:edit", contact.get("id"));
  });

  // edited for brevity
```

assets/js/apps/contacts/show/show_view.js

```
Show.Contact = Marionette.LayoutView.extend({
  // edited for brevity

  events: {
    "click a.js-edit": "editClicked"
  },

  onShow: function(){
    var contact = this.model;
    var acquaintances = contact.get("acquaintances");
    var strangers = contact.get("strangers");

    var acquaintancesView = new ContactManager.Common.Views.PaginatedView({
      collection: acquaintances,
      mainView: Show.Acquaintances,
      propagatedEvents: ["childview:acquaintance:remove"]
    });

    var strangersView = new ContactManager.Common.Views.PaginatedView({
      collection: strangers,
      mainView: Show.Strangers,
      propagatedEvents: ["childview:acquaintance:add"]
```

```
23      });
24
25      this.listenTo(acquaintancesView, "childview:acquaintance:remove",
26                                                          function(view, args){
27        contact.get("acquaintances").remove(args.model);
28      });
29      this.listenTo(strangersView, "childview:acquaintance:add",
30                                                          function(view, args){
31        contact.get("acquaintances").add(args.model);
32      });
33
34      this.acquaintancesRegion.show(acquaintancesView);
35      this.strangersRegion.show(strangersView);
36    }
37  });
```

Don't forget the `contact` variable declaration on line 9!

To simplify our controller further, let's delegate the responsibility of fetching related data (i.e. acquaintances and strangers) to the view:

assets/js/apps/contacts/show/show_controller.js

```
1   $.when(fetchingContact).done(function(contact){
2   ~~  var acquaintancesFetched = contact.get("acquaintances").fetch({silent: true}),~~
3   ~~      strangersFetched = contact.get("strangers").fetch({silent: true});~~
4   ~~  $.when(acquaintancesFetched, strangersFetched).done(function(){~~
5   ~~    var contactView = new Show.Contact({~~
6   ~~      model: contact~~
7   ~~    });~~
8
9   ~~    Show.Controller.listenTo(contactView, "contact:edit", function(contact){~~
10  ~~      ContactManager.trigger("contact:edit", contact.get("id"));~~
11  ~~    });~~
12
13  ~~    ContactManager.mainRegion.show(contactView);~~
14  ~~  });~~
15
16    var contactView = new Show.Contact({
17      model: contact
```

```
  });

  Show.Controller.listenTo(contactView, "contact:edit", function(contact){
    ContactManager.trigger("contact:edit", contact.get("id"));
  });

  ContactManager.mainRegion.show(contactView);
}).fail(function(response){

  // edited for brevity
```

assets/js/apps/contacts/show/show_view.js

```
onShow: function(){
  // edited for brevity

  this.listenTo(acquaintancesView, "childview:acquaintance:remove",
                                                 function(view, args){
    contact.get("acquaintances").remove(args.model);
  });
  this.listenTo(strangersView, "childview:acquaintance:add", function(view, args\
){
    contact.get("acquaintances").add(args.model);
  });

  this.acquaintancesRegion.show(acquaintancesView);
  this.strangersRegion.show(strangersView);
  var acquaintancesFetched = contact.get("acquaintances").fetch({silent: true}),
      strangersFetched = contact.get("strangers").fetch({silent: true});
  var self = this;
  $.when(acquaintancesFetched, strangersFetched).done(function(){
    self.acquaintancesRegion.show(acquaintancesView);
    self.strangersRegion.show(strangersView);
  });
}
```

Git commit using Backbone.associations to handle sub-model relationships:

16fe720ffa962f319cb40617c1f654452634943c[117]

[117] https://github.com/davidsulc/marionette-serious-progression-16fe720ffa962f319cb40617c1f654452634943c

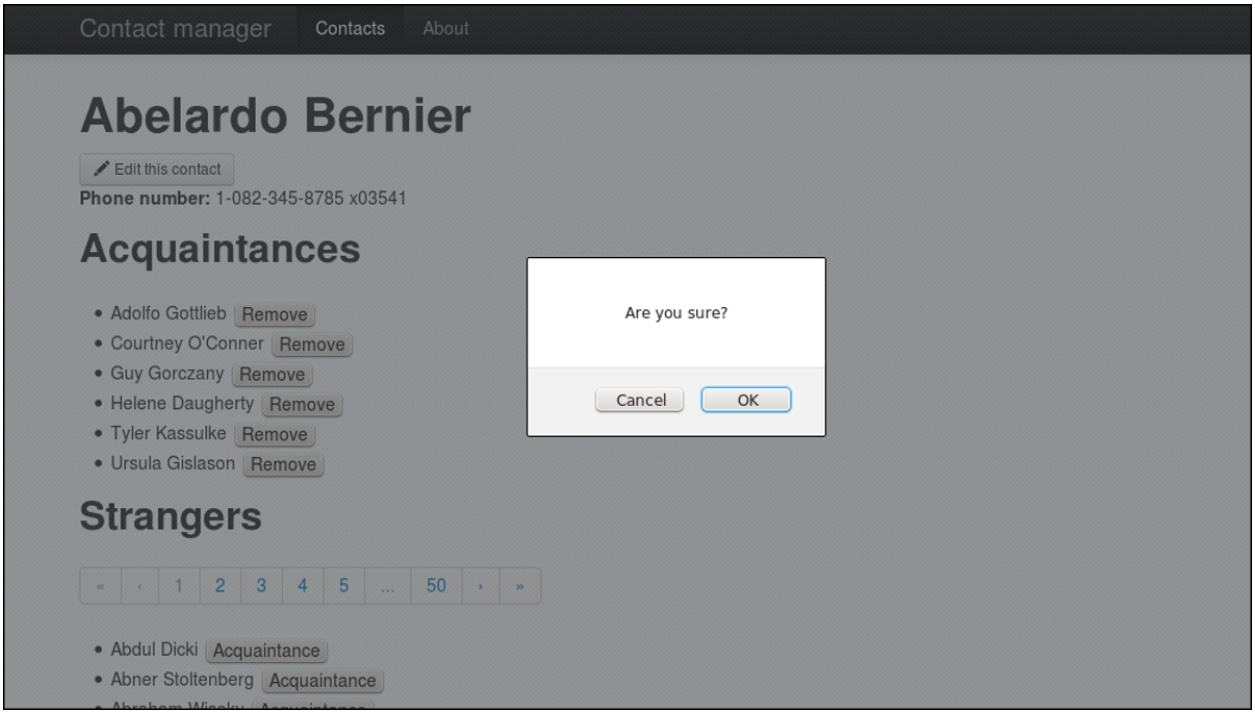
Marionette Behaviors

We should probably ask our users to confirm when they want to add a contact to another contact's list of acquaintances. Let's do just that:

assets/js/apps/contacts/show/show_view.js

```
Show.Stranger = Marionette.ItemView.extend({
  tagName: "li",
  template: "#contact-stranger-view",
  triggers: {
    "click .js-add-acquaintance": "acquaintance:add"
  }
  events: {
    "click .js-add-acquaintance": "addAcquaintance"
  },

  addAcquaintance: function(e){
    if(confirm("Are you sure?")){
      this.trigger("acquaintance:add", { model: this.model })
    }
  }
});
```

Nothing special is happening here, except for the fact that we're providing an object as the trigger parameter to remain compatible with our trigger listeners. Since this event was previously triggered by a `triggers` hash (which automatically provides an object with the event's context), we're simply reproducing that (although we're only providing the model, because that's all our listeners want to access). Here's what the result looks like:



The confirmation request displayed

Great, now let's also ask for confirmation when the user wants to *remove* an acquaintance. But let's think about this for a second before we start copying and pasting all over the place. What is the behavior we're looking to implement? We just want to ask for confirmation when a user clicks on a button, and trigger an event if the action is confirmed, right? That sounds pretty generic, so maybe we should abstract that behavior in our app.

Introducing Marionette Behaviors

Marionette Behaviors[118] are meant to do just that: extract common behavior so the views only define configuration details. Let's start by defining a `Confirmable` behavior in a new file:

assets/js/common/behaviors.js

```
1  ContactManager.Behaviors = {
2    Confirmable: Marionette.Behavior.extend({
3      events: {
4        "click .js-behavior-confirmable": "confirmAction"
5      },
6
7      confirmAction: function() {
8        if(confirm("Are you sure?")){
```

[118] https://github.com/marionettejs/backbone.marionette/blob/master/docs/marionette.behavior.md

```
9           this.view.trigger(this.options.event, { model: this.view.model });
10        }
11      }
12    })
13  };
14
15  Marionette.Behaviors.behaviorsLookup = function(){
16    return ContactManager.Behaviors;
17  }
```

So, what are we up to here? On line 1, we define an object to hold all of our behaviors, and we define our `Confirmable` behavior on lines 2-12. We then tell Marionette where to locate our behavior definitions on lines 15-17.

Within the behavior definition itself, we essentially have the same code we'd put within the view. But since we're *not* within a view, Marionette provides access to a few handy attributes, such as the `view` we use on line 9, referring to the view in which the behavior is included. You can see more on the various attributes that behaviors give you access to in the documentation[119].

We've changed our selection class to "js-behavior-confirmable" instead of "js-add-acquaintance": since behaviors are meant to be generic, the selectors should be, too.

Naturally, we'll need to include this new file:

index.html

```
1  <!-- edited for brevity -->
2  <script src="./assets/js/common/views.js"></script>
3  <script src="./assets/js/common/behaviors.js"></script>
4
5  <script src="./assets/js/apps/contacts/contacts_app.js"></script>
6  <script src="./assets/js/apps/contacts/common/views.js"></script>
7  <!-- edited for brevity -->
```

And we need to update our template to use the proper class selector:

[119] https://github.com/marionettejs/backbone.marionette/blob/master/docs/marionette.behavior.md

index.html

```
<script type="text/template" id="contact-stranger-view">
  {{- fullName }} <button class="js-add-acquaintance">Acquaintance</button>
  {{- fullName }} <button class="js-behavior-confirmable">Acquaintance</button>
</script>
```

Last, but certainly not least, we need to configure the behavior in our view:

assets/js/apps/contacts/show/show_view.js

```
Show.Stranger = Marionette.ItemView.extend({
  tagName: "li",
  template: "#contact-stranger-view",
  events: {
    "click .js-add-acquaintance": "addAcquaintance"
  },

  addAcquaintance: function(e){
    if(confirm("Are you sure?")){
      this.trigger("acquaintance:add", { model: this.model })
    }
  }

  behaviors: {
    Confirmable: {
      event: "acquaintance:add"
    }
  }
});
```

The confirmation dialog is now displayed using behaviors, but we can still improve on it: let's add the possibility to add custom messages, with a default value. We'll start by adding the default message:

assets/js/common/behaviors.js

```
ContactManager.Behaviors = {
  Confirmable: Marionette.Behavior.extend({
    defaults: {
      message: "Are you sure?"
    },
    events: {
      "click .js-behavior-confirmable": "confirmAction"
    },

    confirmAction: function() {
      if(confirm(this.options.message)){
        this.view.trigger(this.options.event, { model: this.view.model });
      }
    }
  })
};
```

This change makes our behavior expect a `message` value in the configuration options, but fall back to a default "Are you sure?" message. If you try adding an acquaintance again, you'll see exactly the same confirmation dialog as above. We can now easily change the confirmation message from our view:

assets/js/apps/contacts/show/show_view.js

```
Show.Stranger = Marionette.ItemView.extend({
  tagName: "li",
  template: "#contact-stranger-view",

  behaviors: {
    Confirmable: {
      event: "acquaintance:add",
      message: "Add this person as an acquaintance?"
    }
  }
});
```

Let's now try and make this message more personal, by asking "Add Mike as an acquaintance?" if the user clicks on a contact whose first name is Mike. But since we're defining behaviors in the view's definition, we don't have access to the view's `model` attribute... How can we solve this issue? Well, we'll turn our `message` attribute into a function that will be provided the behavior's `view` attribute. Here we go:

assets/js/apps/contacts/show/show_view.js

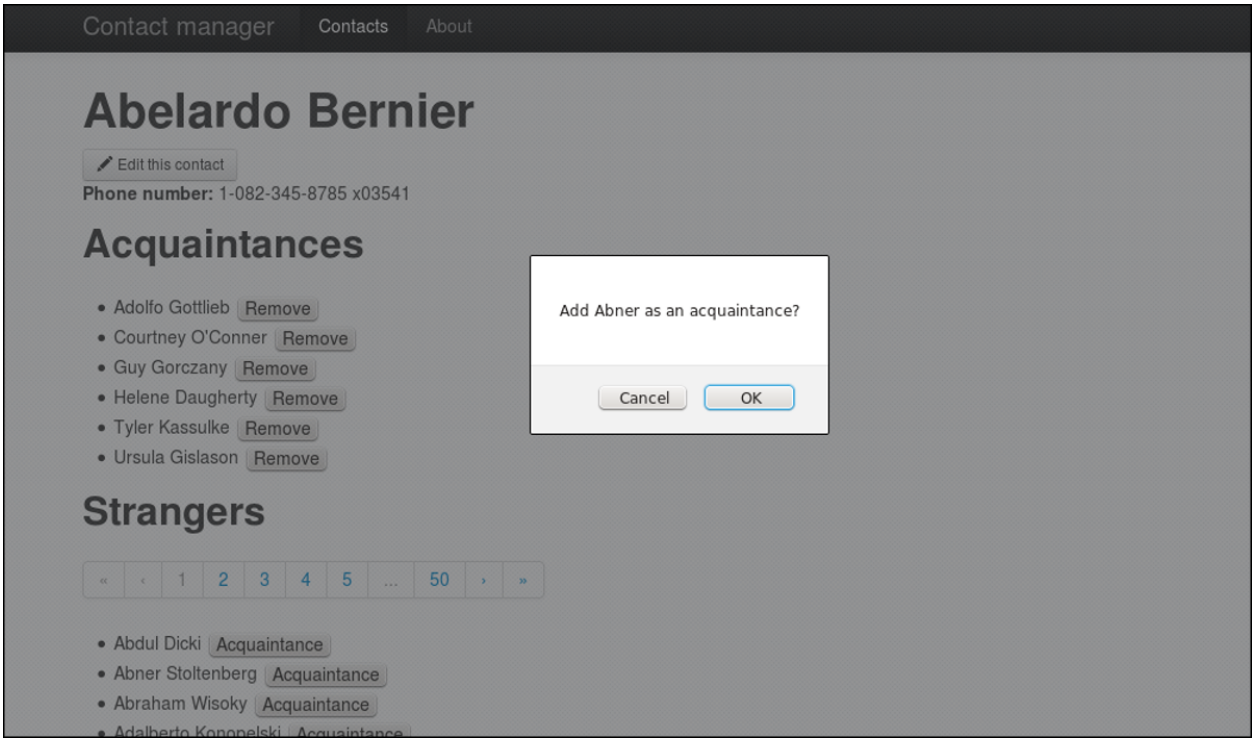
```
behaviors: {
  Confirmable: {
    event: "acquaintance:add",
    message: function(view){
      return "Add " + view.model.get("firstName") + " as an acquaintance?";
    }
  }
}
```

Of course, we need to update our behavior to be able to process confirmation messages as both strings and functions:

assets/js/common/behaviors.js

```
ContactManager.Behaviors = {
  Confirmable: Marionette.Behavior.extend({
    defaults: {
      message: "Are you sure?"
    },
    events: {
      "click .js-behavior-confirmable": "confirmAction"
    },

    confirmAction: function() {
      var message = this.options.message;
      if(typeof(this.options.message) === "function"){
        message = this.options.message(this.view);
      }
      if(confirm(message)){
      if(confirm(this.options.message)){
        this.view.trigger(this.options.event, { model: this.view.model });
      }
    }
  })
};
```

And here's the result of our efforts:



Since we've done most of the work, let's add a basic confirmation request when removing acquaintances:

assets/js/apps/contacts/show/show_view.js

```
1  Show.Acquaintance = Marionette.ItemView.extend({
2    tagName: "li",
3    template: "#contact-acquaintance-view",
4  - triggers: {
5  -   "click .js-remove-acquaintance": "acquaintance:"
6  - }
7
8    behaviors: {
9      Confirmable: {
10       event: "acquaintance:remove"
11     }
12   }
13 });
```

And we'll have to update our template, so our behavior event has the proper class selector present:

index.html

```
<script type="text/template" id="contact-acquaintance-view">
  {{- fullName }} <button class="js-remove-acquaintance">Remove</button>
  {{- fullName }} <button class="js-behavior-confirmable">Remove</button>
</script>
```

Now that we're officially on a roll, try adding the following functionality yourself: on the page listing all contacts, only delete a contact if the user confirms the action in the dialog. Then, take a look at the next page for the solution.

We simply need to add a behavior in the proper view:

assets/js/apps/contacts/list/list_view.js

```
List.Contact = Marionette.ItemView.extend({
  tagName: "tr",
  template: "#contact-list-item",

  triggers: {
    "click td a.js-show": "contact:show",
    "click td a.js-edit": "contact:edit"
    "click td a.js-edit": "contact:edit",
    "click button.js-delete": "contact:delete"
  },

  events: {
    "click": "highlightName"
  },

  modelEvents: {
    "change": "render"
  },

  behaviors: {
    Confirmable: {
      event: "contact:delete"
    }
  }
});
```

And update our template:

index.html

```
<script type="text/template" id="contact-list-item">
  <td>{{- firstName }}</td>
  <td>{{- lastName }}</td>
  <td>
    <a href="#contacts/{{- id }}" class="btn btn-small js-show">
      <i class="icon-eye-open"></i>
      Show
    </a>
    <a href="#contacts/{{- id }}/edit" class="btn btn-small js-edit">
```

```
      <i class="icon-pencil"></i>
      Edit
    </a>
    <button class="btn btn-small js-delete">
    <button class="btn btn-small js-behavior-confirmable">
      <i class="icon-remove"></i>
      Delete
    </button>
  </td>
</script>
```

In some cases, you might want to have a callback executed by your behavior. This is quite easy to implement: you just need to pass the callback as an option and execute it when appropriate. For example:

assets/js/common/behaviors.js

```
confirmAction: function() {
  var message = this.options.message;
  if(typeof(this.options.message) === "function"){
    message = this.options.message(this.view);
  }
  if(confirm(message)){
    this.view.trigger(this.options.event, { model: this.view.model });
    if(this.options.callback){
      this.options.callback();
    }
  }
}
```

We can then specify a callback when configuring our behavior:

assets/js/apps/contacts/list/list_view.js

```
behaviors: {
  Confirmable: {
    event: "contact:delete",
    callback: function(){ console.log("Delete confirmed!"); }
  }
}
```

Git commit implementing behaviors

56494e3a43f00a6733a4d50805b4dfb9adde06d6[120]

[120] https://github.com/davidsulc/marionette-serious-progression-56494e3a43f00a6733a4d50805b4dfb9adde06d6

Internationalization

On the last leg of our journey, we'll cover internationalization (also known by its "i18n" contraction because it starts with "i", ends with "n", and had has 18 letters in between...) by translating our application into French, using polyglot[121].

Managing Routes

We'll want to have our current language appear in the URL, so that means we'll need to update our route management code. Assuming we keep track of the current language within our app (in a `currentLanguage` attribute), let's start by adding that language to the route when we navigate:

assets/js/app.js

```
1  ContactManager.navigate = function(route,  options){
2    options || (options = {});
3    route = ContactManager.i18n.currentLanguage + "/" + route;
4    Backbone.history.navigate(route, options);
5  };
```

Naturally, having languages in the routes means we need to process them correctly:

assets/js/apps/about/about_app.js

```
1  AboutAppRouter.Router = Marionette.AppRouter.extend({
2    appRoutes: {
3      "about" : "showAbout"
4      ":lang/about" : "showAbout"
5    }
6  });
7
8  var API = {
9    showAbout: function(){
10   showAbout: function(lang){
11     ContactManager.setLanguage(lang);
12
13     // edited for brevity
```

And of course, we need to define the `setLanguage` function we use on line 11 above:

[121] http://airbnb.github.io/polyglot.js/

assets/js/app.js

```
ContactManager.setLanguage = function(lang){
  if(ContactManager.i18n.acceptedLanguages.indexOf(lang) > -1){
    if(ContactManager.i18n.currentLanguage !== lang){
      console.log("Language has changed to '" + lang + "'.");
    }
    ContactManager.i18n.currentLanguage = lang;
  }
  return ContactManager.i18n.currentLanguage;
};
```

For handling the languages within the "contacts" sub-app, we'll use a different strategy. Instead of calling a method to set the language for each API action, we can leverage Backbone's `execute` method. From the documentation[122]:

> This method is called internally within the router, whenever a route matches and its corresponding callback is about to be executed. Override it to perform custom parsing or wrapping of your routes, for example.

In other words, in addition to setting the current language, we can use `execute` to replace our `parseParams` function:

assets/js/apps/contacts/contacts_app.js

```
ContactsAppRouter.Router = Marionette.AppRouter.extend({
  appRoutes: {
    ~~":lang/contacts(/filter/:params)": "listContacts",~~
    ~~":lang/contacts/:id": "showContact",~~
    ~~":lang/contacts/:id/edit": "editContact"~~
  ~~},~~
    ":lang/contacts(/filter/:params)": "listContacts",
    ":lang/contacts/:id": "showContact",
    ":lang/contacts/:id/edit": "editContact"
  },

  ~~var parseParams = function(params){~~
    ~~var options = {};~~
    ~~if(params && params.trim() !== ''){~~
      ~~params = params.split('+');~~
```

[122] http://backbonejs.org/#Router-execute

```
      _.each(params, function(param){
        var values = param.split(':');
        if(values[1]){
          if(values[0] === "page"){
            options[values[0]] = parseInt(values[1], 10);
          }
          else{
            options[values[0]] = values[1];
          }
        }
      });
    }
    _.defaults(options, { page: 1 });
    return options;
  };

  execute: function(callback, args){
    var lang = args[0],
        params = args[1],
        query = args[2] || '',
        options = {};

    ContactManager.setLanguage(lang);

    if(params){
      if(params.indexOf(":") < 0){
        options = params;
      }
      else{
        if(params.trim() !== ''){
          params = params.split('+');
          _.each(params, function(param){
            var values = param.split(':');
            if(values[1]){
              if(values[0] === "page"){
                options[values[0]] = parseInt(values[1], 10);
              }
              else{
                options[values[0]] = values[1];
              }
            }
          });
```

```
58          }
59        }
60      }
61
62      _.defaults(options, { page: 1 });
63      if(callback){
64        callback.call(this, options);
65      }
66    }
67  });
```

As you can see on lines 33-36, `execute` gets passed arguments: the various arguments from our route, as well as the query string (if any). In other words, you could use `execute` to parse the query string in your Backbone/Marionette app and use values for client side configuration (for example).

Note that on line 40, we need to explicitly check for the presence of parameters, because the root route will call `execute` without any parameters: in that case, we still need to route our app to the default page. Also, the call to `defaults` on line 62 will only add a `page` key if `options` is an object: in the case where `options` is (e.g.) just a simple id (as is the case when calling "show"), the call to `defaults` can basically be ignored. Then, on lines 63-65, we call the provided callback (if applicable), passing it the options object.

Let's also update our `listContacts` function:

assets/js/apps/contacts/contacts_app.js

```
1  var API = {
2    listContacts: function(params){
3      var options = parseParams(params);
4    listContacts: function(options){
5      options || (options = { page: 1 });
6      executeAction(ContactManager.ContactsApp.List.Controller.listContacts,
7                                                                    options);
8    },
```

Of course, we'll only want to handle a short list of languages in our app, so let's define that list when we start our app:

index.html

```
<script type="text/javascript">
  ContactManager.start();
  ContactManager.start({
    acceptedLanguages: ["en", "fr"]
  });
</script>
```

And we'll need to store these languages within the app, so we can refer to them later:

assets/js/app.js

```
ContactManager.on("start", function(){
ContactManager.on("start", function(options){
  options || (options = {});
  ContactManager.i18n = {
    acceptedLanguages: options.acceptedLanguages || [],
    currentLanguage: "en"
  };

  _.templateSettings = {

    // edited for brevity
```

Redirecting Unsupported Languages

Now that we've defined which languages our app will support, we'll need to redirect any requests attempting to change the current language to an unsupported value. These requests will be redirected to a default language:

assets/js/app.js

```
ContactManager.setLanguage = function(lang){
  if(ContactManager.i18n.acceptedLanguages.indexOf(lang) > -1){
    if(ContactManager.i18n.currentLanguage !== lang){
      console.log("Language has changed to '" + lang + "'.");
    }
    ContactManager.i18n.currentLanguage = lang;
  }
  else{
    var currentRoute = ContactManager.getCurrentRoute().split("/");
```

```
10      currentRoute.shift();
11      ContactManager.navigate(currentRoute.join("/"));
12    }
13    return ContactManager.i18n.currentLanguage;
14  };
```

Notice that one lines 9-11 we basically take out the language parameter from the route, and redirect to the same route (since our `navigate` method will add the current language back into the URL).

Internationalizing Links

Now that we're processing the language in the URL, we need to generate links in our views containing the current language. To achieve that, we'll use template helpers[123], which are functions that Marionette makes accessible within templates. Let's start with listing contacts:

assets/js/apps/contacts/list/list_view.js

```
1  List.Contact = Marionette.ItemView.extend({
2    tagName: "tr",
3    template: "#contact-list-item",
4
5    templateHelpers: {
6      i18nUrl: function(url){
7        return ContactManager.i18n.currentLanguage + "/" + url;
8      }
9    },
10
11   // edited for brevity
```

index.html

```
1  <script type="text/template" id="contact-list-item">
2    <td>{{- firstName }}</td>
3    <td>{{- lastName }}</td>
4    <td>
5      <a href="#contacts/{{- id }}" class="btn btn-small js-show">
6      <a href="#{{- i18nUrl('contact/' + id) }}" class="btn btn-small js-show">
7        <i class="icon-eye-open"></i>
8        Show
```

[123] https://github.com/marionettejs/backbone.marionette/blob/master/docs/marionette.view.md#viewtemplatehelpers

```
    </a>
    <a href="#contacts/{{- id }}/edit" class="btn btn-small js-show">
    <a href="#{{- i18nUrl('contact/' + id + '/edit') }}"
                                          class="btn btn-small js-edit">
      <i class="icon-pencil"></i>
      Edit
    </a>

    <!-- edited for brevity -->
```

Pretty easy, right? We simply prefix the language to the URL fragment we want to navigate to. Let's do this in all of our templates, which means we'll need to have our trusty `i18nUrl` template helper accessible in all of our views. Instead of defining the template helper in every view, we'll extend Marionette's base view (in a new file):

assets/js/apps/config/marionette/views/template_helpers.js

```
_.extend(Marionette.View.prototype, {
  templateHelpers: {
    i18nUrl: function(url){
      return ContactManager.i18n.currentLanguage + "/" + url;
    }
  }
});
```

Naturally, we'll no longer need the same function in our "list" view:

assets/js/apps/contacts/list/list_view.js

```
List.Contact = Marionette.ItemView.extend({
  tagName: "tr",
  template: "#contact-list-item",

  templateHelpers: {
    i18nUrl: function(url){
      return ContactManager.i18n.currentLanguage + "/" + url;
    }
  },

  // edited for brevity
```

And we'll need to include our new file within our app:

index.html

```
<!-- edited for brevity -->
<script src="./assets/js/vendor/spin.jquery.js"></script>

<script src="./assets/js/apps/config/marionette/views/template_helpers.js">
                                                                    </script>
<script src="./assets/js/apps/config/marionette/regions/dialog.js"></script>
<!-- edited for brevity -->
```

Now, we can update the links we use within our app:

index.html

```
<!-- edited for brevity -->

<script type="text/template" id="header-template">
  <div class="navbar-inner">
    <div class="container">
      <a class="brand" href="#contacts">Contact manager</a>
      <a class="brand" href="#{{= i18nUrl('contacts') }}">Contact manager</a>
      <div class="nav-collapse collapse">
        <ul class="nav"></ul>
      </div>
    </div>
  </div>
</script>

<!-- edited for brevity -->

<script type="text/template" id="header-link">
  <a href="#{{- url }}">{{- name }}</a>
  <a href="#{{- i18nUrl(url) }}">{{- name }}</a>
</script>

<!-- edited for brevity -->

<script type="text/template" id="contact-view">
  <h1>{{- fullName }}</h1>
  <a href="#contacts/{{- id }}/edit" class="btn btn-small js-edit">
  <a href="#{{- i18nUrl('contacts/' + id + '/edit') }}"
                                    class="btn btn-small js-edit">
    <i class="icon-pencil"></i>
```

```
    Edit this contact
  </a>

<!-- edited for brevity -->

<script type="text/template" id="pagination-controls">
  {{ if(totalPages > 1){ }}
    <ul>
      {{ if(currentPage > 1){ }}
        <li><a href="#{{- urlBase ? urlBase + 1 : "" }}" class="navigatable"
                                                 data-page="1">&laquo;</a></li>
        <li><a href="#{{- urlBase ? urlBase + previous  : "" }}"
            class="navigatable" data-page="{{- previous }}">&lsaquo;</a></li>
        <li><a href="#{{- urlBase ? i18nUrl(urlBase) + 1 : "" }}"
                             class="navigatable" data-page="1">&laquo;</a></li>
        <li><a href="#{{- urlBase ? i18nUrl(urlBase) + previous  : "" }}"
            class="navigatable" data-page="{{- previous }}">&lsaquo;</a></li>
      {{ }else{ }}

      <!-- edited for brevity -->

        {{ }else{ }}
          <li><a href="#{{- urlBase ? urlBase + page  : "" }}"
            class="navigatable" data-page="{{- page }}">{{- page }}</a></li>
          <li><a href="#{{- urlBase ? i18nUrl(urlBase) + page  : "" }}"
            class="navigatable" data-page="{{- page }}">{{- page }}</a></li>
        {{ } }}
      {{ }); }}

      {{ if(pageSet[pageSet.length - 1] !== lastPage){ }}
        <li class="disabled"><a href="#">...</a></li>
        <li><a href="#{{- urlBase ? urlBase + lastPage  : "" }}"
            class="navigatable" data-page="{{- lastPage }}">
                                                 {{- lastPage }}</a></li>
        <li><a href="#{{- urlBase ? i18nUrl(urlBase) + lastPage  : "" }}"
            class="navigatable" data-page="{{- lastPage }}">
                                                 {{- lastPage }}</a></li>
      {{ } }}

      {{ if(currentPage !== lastPage){ }}
        <li><a href="#{{- urlBase ? urlBase + next  : "" }}"
            class="navigatable" data-page="{{- next }}">&rsaquo;</a></li>
```

```
        <li><a href="#{{- urlBase ? urlBase + lastPage  : "" }}"
         class="navigatable" data-page="{{- lastPage }}">&raquo;</a></li>
        <li><a href="#{{- urlBase ? i18nUrl(urlBase) + next  : "" }}"
            class="navigatable" data-page="{{- next }}">&rsaquo;</a></li>
        <li><a href="#{{- urlBase ? i18nUrl(urlBase) + lastPage  : "" }}"
            class="navigatable" data-page="{{- lastPage }}">&raquo;</a></li>
      {{ }else{ }}
        <li class="disabled"><a href="#">&rsaquo;</a></li>
        <li class="disabled"><a href="#">&raquo;</a></li>
      {{ } }}
    </ul>
  {{ } }}
</script>
```

Let's improve our app's design by using an event to signal the language has changed. That way, once we let users change the language with a dropdown, we can have our app listen for that event and translate our application (loading the new translation data, etc.). We'll also use this event to make our header rerender itself, so the links update with the new language value.

Implementing the event (assets/js/app.js)

```
ContactManager.setLanguage = function(lang){
  if(ContactManager.i18n.acceptedLanguages.indexOf(lang) > -1){
    if(ContactManager.i18n.currentLanguage !== lang){
      console.log("Language has changed to '" + lang + "'.");
      ContactManager.trigger("language:changed", lang);
    }
    ContactManager.i18n.currentLanguage = lang;
  }
  // edited for brevity
```

Listening for the event (assets/js/app.js)

```
ContactManager.on("start", function(){
  // edited for brevity
});

ContactManager.on("language:changed", function(lang){
  console.log("Language has changed to '" + lang + "'.");
  ContactManager.i18n.currentLanguage = lang;
});
```

Rerendering headers on language change (assets/js/apps/header/header_app.js)

```
ContactManager.module("HeaderApp", function(Header, ContactManager,
                                             Backbone, Marionette, $, _){
  // edited for brevity

  this.listenTo(Header, "start", function(){
    API.listHeader();
  });

  this.listenTo(ContactManager, "language:changed", function(){
    API.listHeader();
  });
});
```

With this in place, if we execute

```
ContactManager.trigger("language:changed", "fr");
```

from the console and then hover over the header links, we can see their language parameter has been correctly updated.

Introducing Polyglot

Now that our routes are more or less taken care of, we can move on to actual translation. Start by downloading the library from here[124] and saving it in *assets/js/vendor/polyglot.js*.

Including the polyglot library (index.html)

```
<!-- edited for brevity -->
<script src="./assets/js/vendor/backbone.marionette.js"></script>
<script src="./assets/js/vendor/polyglot.js"></script>
<script src="./assets/js/vendor/spin.js"></script>
<script src="./assets/js/vendor/spin.jquery.js"></script>
<!-- edited for brevity -->
```

Let's try polyglot out in the console, to get a feel for how it works:

[124] https://raw.githubusercontent.com/davidsulc/marionette-serious-progression-app/master/assets/js/vendor/polyglot.js

Console interactions

```
1  var polyglot = new Polyglot();
2  polyglot.extend({
3    "hello": "Hello"
4  });
5
6  console.log(polyglot.t("hello")); // "Hello"
7
8  polyglot.extend({
9    "hello": "Bonjour"
10 });
11
12 console.log(polyglot.t("hello")); // "Bonjour"
```

As you can see, it's pretty simple to use: after instantiating polyglot (line 1), simply use the `extend` method to provide a hash of key-translation data (lines 2-4). Then, you call the `t` method to get the translation corresponding to a key (line 6). You can call `extend` again to switch languages (lines 8-10).

Instead of calling `polyglot.t` everywhere, we call alias this to a global `t` function to handle all translation. Let's see how that works:

Console interactions

```
1  var polyglot = new Polyglot();
2  _.bindAll(polyglot, "t");
3  window.t = polyglot.t;
4
5  polyglot.extend({
6    "hello": "Hello",
7    "contact.firstName": "First Name"
8  });
9
10 console.log(t("hello")); // "Hello"
11 console.log(t("contact.firstName")); // "First Name"
```

On lines 2-3, we use Underscore to bind the `t` function's context to `polyglot`, then alias `polyglot.t` as a global `t` function. On lines 10-11, we call `t` and see that it translates data as expected. You'll also notice that we can use "." to "scope" translation strings and make it easier to see what they refer to.

Translating Templates

Let's use our new `t` function to translate template strings. For starters, we'll need to configure polyglot in our index page: we'll do so right after loading all of the vendor libraries, so that translations will be working when we load our app's files.

index.html

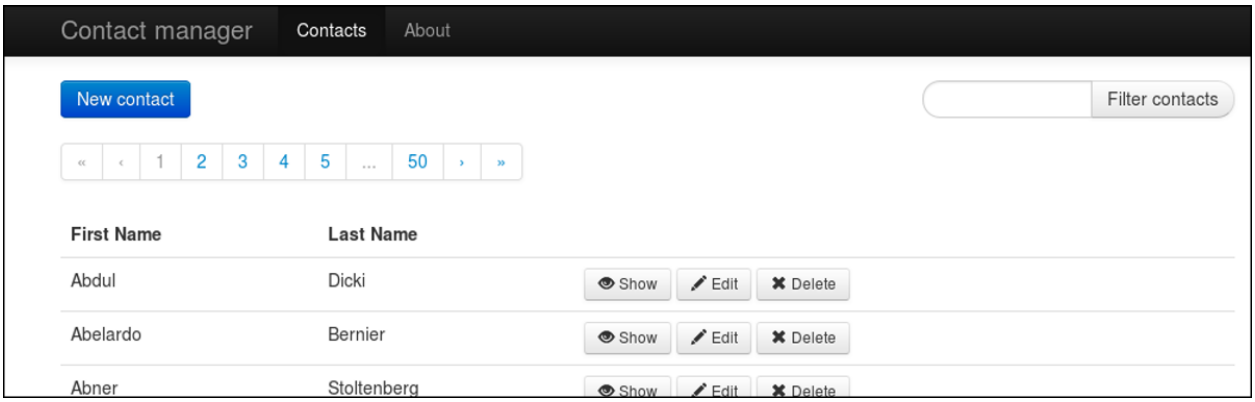
```
<!-- edited for brevity -->
<script src="./assets/js/vendor/backbone.marionette.js"></script>
<script src="./assets/js/vendor/polyglot.js"></script>
<script src="./assets/js/vendor/spin.js"></script>
<script src="./assets/js/vendor/spin.jquery.js"></script>

<script type="text/javascript">
  var polyglot = new Polyglot();
  _.bindAll(polyglot, "t");
  window.t = polyglot.t;

  polyglot.extend({
    "contact.firstName": "First Name",
    "contact.lastName": "Last Name"
  });
</script>

<script src="./assets/js/apps/config/marionette/views/template_helpers.js">
                                                                    </script>
<!-- edited for brevity -->
```

With polyglot configured, we simply need to call `t` in our templates:

index.html

```
<script type="text/template" id="contact-list">
  <thead>
    <tr>
      <th>First Name</th>
      <th>Last Name</th>
      <th>{{= t("contact.firstName") }}</th>
      <th>{{= t("contact.lastName") }}</th>
      <th></th>
    </tr>
  </thead>
```

```
11    <tbody>
12    </tbody>
13  </script>
```

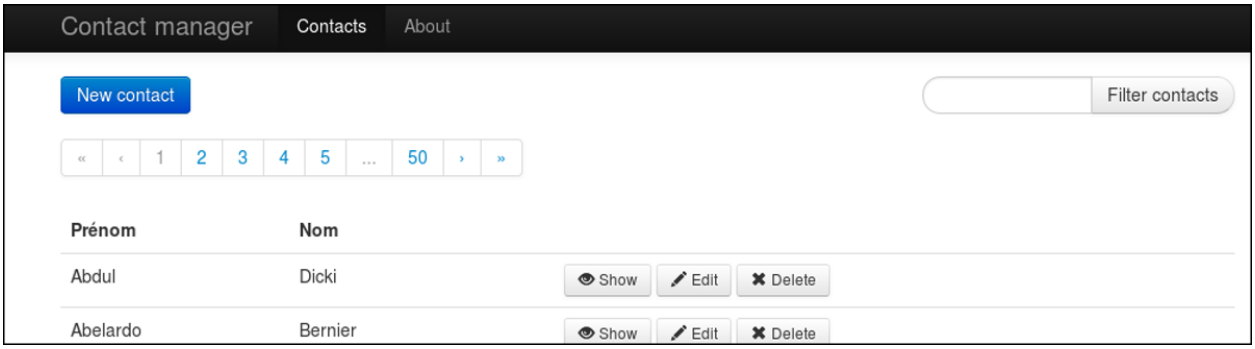


Our headers displayed with polyglot: still the same!

And now, let's load some French translations and see what we get:

index.html

```
1   <script type="text/javascript">
2     var polyglot = new Polyglot();
3     _.bindAll(polyglot, "t");
4     window.t = polyglot.t;
5
6     polyglot.extend({
7       "contact.firstName": "First Name",
8       "contact.lastName": "Last Name"
9     });
10
11    polyglot.extend({
12      "contact.firstName": "Prénom",
13      "contact.lastName": "Nom"
14    });
15  </script>
```



Our headers translated by polyglot

We can now simply add all of our translations:

index.html

```
polyglot.extend({
  "generic.show": "Show",
  "generic.edit": "Edit",
  "generic.save": "Save",
  "generic.delete": "Delete",
  "generic.remove": "Remove",
  "generic.changedOnServer": "This model has changed on the server, and has been
                              updated with the latest data from the server and
                              your changes have been reapplied.",
  "generic.confirmationMessage": "Are you sure?",
  "generic.unprocessedError": "An unprocessed error happened. Please try again!",

  "menu.Contacts": "Contacts",
  "menu.About": "About",

  "about.title": "About this application",
  "about.message.design": "This application was designed to accompany you during
                           your learning.",
  "about.message.closing": "Hopefully, it has served you well !",

  "acquaintance.modelName": "Acquaintance",
  "acquaintance.addConfirmation": "Add %{firstName} as an acquaintance?",

  "contact.attributes.firstName": "First Name",
  "contact.attributes.lastName": "Last Name",
  "contact.attributes.phoneNumber": "Phone number",
  "contact.attributes.acquaintances": "Acquaintances",
  "contact.attributes.strangers": "Strangers",

```

```
30    "contact.newContact": "New contact",
31    "contact.editContact": "Edit this contact",
32    "contact.filterContacts": "Filter contacts",
33    "contact.noneToDisplay": "No contacts to display.",
34    "contact.missing": "This contact doesn't exist !",
35
36    "loading.title": "Loading Data",
37    "loading.message": "Please wait, data is loading."
38  });
```

We'll cover the special interpolated translation message on line 22 a little farther down.

Now we simply need to make sure every string we use in our app gets translated, for example:

index.html

```
1   <script type="text/template" id="contact-view">
2     <h1>{{- fullName }}</h1>
3     <a href="#{{- i18nUrl('contacts/' + id + '/edit') }}"
4                                   class="btn btn-small js-edit">
5       <i class="icon-pencil"></i>
6       Edit this contact
7       {{- t("contact.editContact") }}
8     </a>
9     <p><strong>Phone number:</strong> {{- phoneNumber }}</p>
10    <p><strong>{{- t("contact.attributes.phoneNumber") }}:</strong>
11                                                {{- phoneNumber }}</p>
12
13    <h2>Acquaintances</h2>
14    <h2>{{- t("contact.attributes.acquaintances") }}</h2>
15    <div id="acquaintances-region"></div>
16
17    <h2>Strangers</h2>
18    <h2>{{- t("contact.attributes.strangers") }}</h2>
19    <div id="strangers-region"></div>
20  </script>
```

Of course, don't forget the strings in other files:

assets/js/common/behaviors.js

```
ContactManager.Behaviors = {
  Confirmable: Marionette.Behavior.extend({
    defaults: {
      ~~message: "Are you sure?"~~
      message: t("generic.confirmationMessage")
    },
    events: {
      // edited for brevity
```

Check the commit at the end of this chapter to make sure you haven't missed any replacements.

Let's discuss the interpolated translation we have:

```
"acquaintance.addConfirmation": "Add %{firstName} as an acquaintance?",
```

As you may have guessed, this will allow us to provide an argument to the translation (so the argument can be positioned properly within the sentence, according to the current language). Here's how it gets used:

assets/js/apps/contacts/show/show_view.js

```
Confirmable: {
  event: "acquaintance:add",
  message: function(view){
    ~~return "Add " + view.model.get("firstName") + " as an acquaintance?";~~
    return t("acquaintance.addConfirmation", {
      firstName: view.model.get("firstName")
    });
  }
}
```

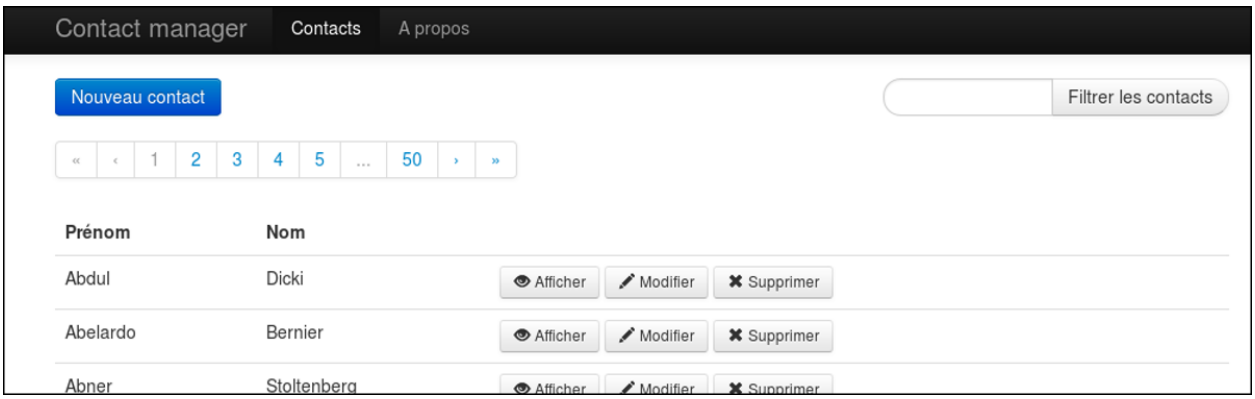
So, now that our app correctly displays messages using polyglot, let's load all the French translations:

index.html

```
  polyglot.extend({
    // English translations
    // edited for brevity
  });

  polyglot.extend({
    "generic.show": "Afficher",
    "generic.edit": "Modifier",
    "generic.save": "Enregistrer",
    "generic.delete": "Supprimer",
    "generic.remove": "Retirer",
    "generic.changedOnServer": "Ce modèle a été modifié sur le serveur, a été mis
                               à jour avec les données les plus récentes
                               provenant du serveur, et vos modifications ont
                               été réappliquées.",
    "generic.confirmationMessage": "Etes-vous sûr ?",
    "generic.unprocessedError": "Une erreur indéterminée est survenue. Veuillez
                               essayer à nouveau !",

    "menu.Contacts": "Contacts",
    "menu.About": "A propos",

    "about.title": "Concernant cette application",
    "about.message.design": "Cette application a été conçue pour vous accompagner
                            au long de votre apprentissage.",
    "about.message.closing": "J'espère qu'elle vous a bien servi !",

    "acquaintance.modelName": "Connaissance",
    "acquaintance.addConfirmation": "Rajouter %{firstName} aux connaissances ?",

    "contact.attributes.firstName": "Prénom",
    "contact.attributes.lastName": "Nom",
    "contact.attributes.phoneNumber": "Numéro de téléphone",
    "contact.attributes.acquaintances": "Connaissances",
    "contact.attributes.strangers": "Inconnus",

    "contact.newContact": "Nouveau contact",
    "contact.editContact": "Modifier ce contact",
    "contact.filterContacts": "Filtrer les contacts",
    "contact.noneToDisplay": "Pas de contacts à afficher",
    "contact.missing": "Ce contact n'existe pas !",
```

```
42
43      "loading.title": "Chargement des données",
44      "loading.message": "Veuillez patienter, les données sont en train d'être
45                          chargées."
46    });
47  </script>
```

Let's take a look at our translated app:



Our app in French

Loading Translation Files

API Properties

We'll use a "languages" endpoint that will provide a hash with the translation keys as above. So the "languages/en" endpoint would return

```
{
    "generic.show": "Afficher",
    "generic.edit": "Modifier",
    "generic.save": "Enregistrer",
    "generic.delete": "Supprimer",

    // edited for brevity

    "contact.missing": "Ce contact n'existe pas !",

    "loading.title": "Chargement des données",
    "loading.message": "Veuillez patienter, les données sont en train d'être
```

```
                        chargées."
}
```

We've demonstrated that our translation works, so let's now get the translation data from the server. Since fetching the translation data from the server will imply some latency, we'll do this using a request:

assets/js/apps/about/about_app.js

```
1  var API = {
2    showAbout: function(lang){
3      ContactManager.setLanguage(lang);
4  —
5      ContactManager.startSubApp("AboutApp");
6      ContactManager.AboutApp.Show.Controller.showAbout();
7      ContactManager.execute("set:active:header", "about");
8      ContactManager.request("language:change", lang).always(function(){
9        ContactManager.startSubApp("AboutApp");
10       ContactManager.AboutApp.Show.Controller.showAbout();
11       ContactManager.execute("set:active:header", "about");
12     });
13   }
14 };
```

Why will we be using a request[125] instead of a command[126]? Because we need to have a promise[127] returned so we can wait until the translations are loaded: commands don't return anything, so they can't be used in this context.

Let's quickly update our contacts app before we go on:

[125] https://github.com/marionettejs/backbone.wreqr#requestresponse
[126] https://github.com/marionettejs/backbone.wreqr#commands
[127] http://api.jquery.com/Types/#Promise

assets/js/apps/contacts/contacts_app.js

```
execute: function(callback, args){
  var lang = args[0],
      params = args[1],
      query = args[2] || '',
      options = {};

  ContactManager.setLanguage(lang);

  ContactManager.request("language:change", lang).always(function(){
    if(params){
      // edited for brevity
    }

    _.defaults(options, { page: 1 });
    if(callback){
      callback.call(this, options);
    }
  });
}
```

});

We can now implement our request handler:

assets/js/app.js

```
ContactManager.setLanguage = function(lang){
  if(ContactManager.i18n.acceptedLanguages.indexOf(lang) > -1){
    if(ContactManager.i18n.currentLanguage !== lang){
      ContactManager.trigger("language:changed", lang);
    }
  }
  else{
    var currentRoute = ContactManager.getCurrentRoute().split("/")
    currentRoute.shift();
    ContactManager.navigate(currentRoute.join("/"));
  }
  return ContactManager.i18n.currentLanguage;
};

ContactManager.startSubApp = function(appName, args){
```

```
16    // edited for brevity{
17  };
18
19  ContactManager.reqres.setHandler("language:change", function(lang){
20    var defer = $.Deferred(),
21        currentRoute = ContactManager.getCurrentRoute().split("/")
22                                                      .slice(1).join("/");
23    if(ContactManager.i18n.acceptedLanguages.indexOf(lang) > -1){
24      if(ContactManager.i18n.currentLanguage !== lang){
25        var translationFetched = $.get("languages/" + lang);
26        $.when(translationFetched).done(function(translation){
27          window.polyglot.extend(translation);
28          ContactManager.i18n.currentLanguage = lang;
29          ContactManager.trigger("language:changed");
30          ContactManager.navigate(currentRoute, {trigger: true})
31          defer.resolve();
32        }).fail(function(){
33          defer.reject();
34          alert(t("contact").generic.unprocessedError);
35        });
36      }
37      else{
38        defer.resolve();
39      }
40    }
41    else{
42      defer.reject();
43      ContactManager.navigate(currentRoute);
44    }
45    return defer.promise();
46  });
47
48  // edited for brevity
49
50  ~~ContactManager.on("language:changed", function(lang){~~
51  ~~  console.log("Language has changed to '" + lang + "'.");~~
52  ~~  ContactManager.i18n.currentLanguage = lang;~~
53  ~~});~~
```

What does this code do? Well, it's very similar to our removed `setLanguage` function: check the requested language is valid, then either update the translations for the new language, or redirect to a supported language. This time around, we strip the language from the current route earlier (lines

21-22), because we'll also need it if the language is valid: after the language has loaded, we navigate the route and explicitly retrigger the route handling code so the app can properly re-initialize itself (i.e. it behaves as if the user left our app, then came back requesting a new language). Note that line 30 is the only case where we navigate with `trigger: true`.

Since line 27 above requires the `polyglot` variable to be global, we need to make a small change:

index.html

```
<script type="text/javascript">
  var polyglot = new Polyglot();
  window.polyglot = new Polyglot();
  _.bindAll(polyglot, "t");
  window.t = polyglot.t;
 // edited for brevity
```

And of course, we only need to have one language's data in our index page:

index.html

```
polyglot.extend({
  "generic.show": "Afficher",
  "generic.edit": "Modifier",

  // edited for brevity

  "loading.title": "Chargement des données",
  "loading.message": "Veuillez patienter, les données sont en train
                      d'être chargées."
});
```

We're now able to load translation data from our server back end. However, we've left the existing `polyglot.extend` call to load the initial translation data: when we first get our index.html page, we can have the server dump the translation data into the HTML, which we can then load into our app. This will save us a request and make our app faster. To be clear, the translation data would normally be dumped into the raw HTML by the server side code (see an example here[128]), *not* copy/pasted in manually.

Another option you have to bootstrap data is dumping the data as JSON, then parsing it client side before providing it to the app:

[128] http://backbonejs.org/#FAQ-bootstrap

index.html

```
<script type="text/translations" id="translations">
  {
    "generic.show": "Show",
    "generic.edit": "Edit",

    // edited for brevity

    "loading.title": "Loading Data",
    "loading.message": "Please wait, data is loading."
  }
</script>

<script type="text/javascript">
  var translation = JSON.parse($("#translations").text());

  window.polyglot = new Polyglot();
  _.bindAll(polyglot, "t");
  window.t = polyglot.t;

  polyglot.extend(translation);
</script>
```

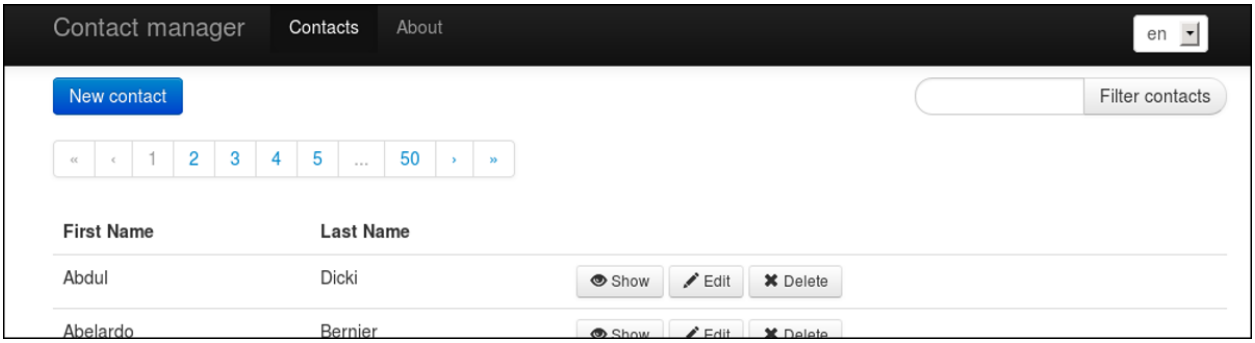
Implementing Language Selection

We've spent a lot of time getting translations to work, but users can't select their language yet. It's time for this grave injustice to come to an end. First, we'll add a dropdown in our menu, providing the language selection from which the user can choose:

index.html

```
<script type="text/template" id="header-template">
  <div class="navbar-inner">
    <div class="container">
      <a class="brand" href="#{{- i18nUrl('contacts') }}">Contact manager</a>
      <div class="nav-collapse collapse">
        <ul class="nav"></ul>
      </div>

      <div class="pull-right" style="padding: 10px 15px;">
        <select class="span1 js-change-language" style="margin-bottom: 0">
```

```
          {{ _.each(ContactManager.i18n.acceptedLanguages, function(lang){ }}
            {{ var selected = ContactManager.i18n.currentLanguage === lang ?
                                                'selected="selected"' : ''; }}
            <option {{- selected }} value="{{- lang }}">{{- lang }}</option>
          {{ }); }}
        </select>
      </div>
    </div>
  </div>
</script>
```



Our language selector

We need to react to the change in the dropdown's value:

assets/js/apps/header/list/list_view.js

```
List.Headers = Marionette.CompositeView.extend({
  // edited for brevity

  events: {
    "click a.brand": "brandClicked",
    "change .js-change-language": "changeLanguage"
  }
});

_.extend(List.Headers.prototype, {
  brandClicked: function(e){
    // edited for brevity
  },

  changeLanguage: function(e){
    e.preventDefault();
    var lang = $(e.target).val();
```

```
18      this.trigger("language:change", lang);
19    }
20  });
```

When that event happens, we need to request a language change:

assets/js/apps/header/list/list_controller.js

```
1   var Controller = Marionette.Controller.extend({
2     listHeader: function(){
3       var links = ContactManager.request("header:entities");
4       var headers = new List.Headers({collection: links});
5
6       // various listeners
7
8       this.listenTo(headers, "language:change", function(lang){
9         ContactManager.request("language:change", lang);
10      });
11
12      ContactManager.headerRegion.show(headers);
13    },
```

And the rest is already wired up for us! We do, however, need to make one small additional change:

assets/js/common/behaviors.js

```
1   Confirmable: Marionette.Behavior.extend({
2     defaults: {
3       message: t("generic.confirmationMessage")
4       message: function(){ return t("generic.confirmationMessage"); }
5     },
6
7     // edited for brevity
```

This is due to the message's dynamic nature: each time it is displayed, it has to be translated into the current language; it can't simply be translated once and left as is, or it won't update when the language changes.

Git commit implementing internationalization:

d4ceae2bce3ad949b908b356b3491aa46a53fbfa[129]

[129] https://github.com/davidsulc/marionette-serious-progression-app/commit/d4ceae2bce3ad949b908b356b3491aa46a53fbfa

Closing Thoughts

We've covered a decent amount of topics in this book, and you should now be comfortable taking on more challenging Marionette implementations. Set your imagination free! With the approaches we've used along our journey, building complex javascript apps shouldn't be too daunting anymore: you now know how to break down complexity and keep it manageable.

If you've enjoyed the book, it would be **immensely helpful** if you could take a few minutes to write your opinion on the book's review page[130]. Help others determine if the book is right for them!

Would you like me to cover another subject in an upcoming book? Let me know by email at davidsulc@gmail.com or on Twitter (@davidsulc). In the meantime, see the next chapter for my current list of books.

Thanks for reading this book, it's been great having you along!

Keeping in Touch

I plan to release more books in the future, where each will teach a new skill (like in this book) or help you improve an existing skill and take it to the next level.

If you'd like to be notified when I release a new book, receive discounts, etc., sign up to my mailing list at davidsulc.com/mailing_list[131]. No spam, I promise!

You can also follow me on Twitter: @davidsulc

[130] https://leanpub.com/marionette-serious-progression/feedback

[131] http://davidsulc.com/mailing_list

Other Books I've Written

Presumably, you're already comfortable with how Marionette works. If that isn't quite the case, take a look the book where the Contact Manager application is originally developed: Backbone.Marionette.js: A Gentle Introduction[132].

If you're interested in adding RequireJS to the mix, check out my next book: Structuring Backbone Code with RequireJS and Marionette Modules[133]. It takes the original Contact Manager application we've started with, and rewrites it to use RequireJS:

- manage dependencies;
- load templates from separate files (templates are no longer in `index.html`!);
- explains typical errors and how you can approach debugging them;
- how to produce a single, optimized, and minifed javascript file of your application that is ready for production.

And best of all, it is written in an exercise style: each chapter introduces what you need to develop, and points out the various things to keep in mind (dependencies, loading order, etc.). You then add the new module to the application by yourself, and can check your answer by reading the step-by-step explanation that follows in the chapter.

[132] https://leanpub.com/marionette-gentle-introduction

[133] https://leanpub.com/structuring-backbone-with-requirejs-and-marionette