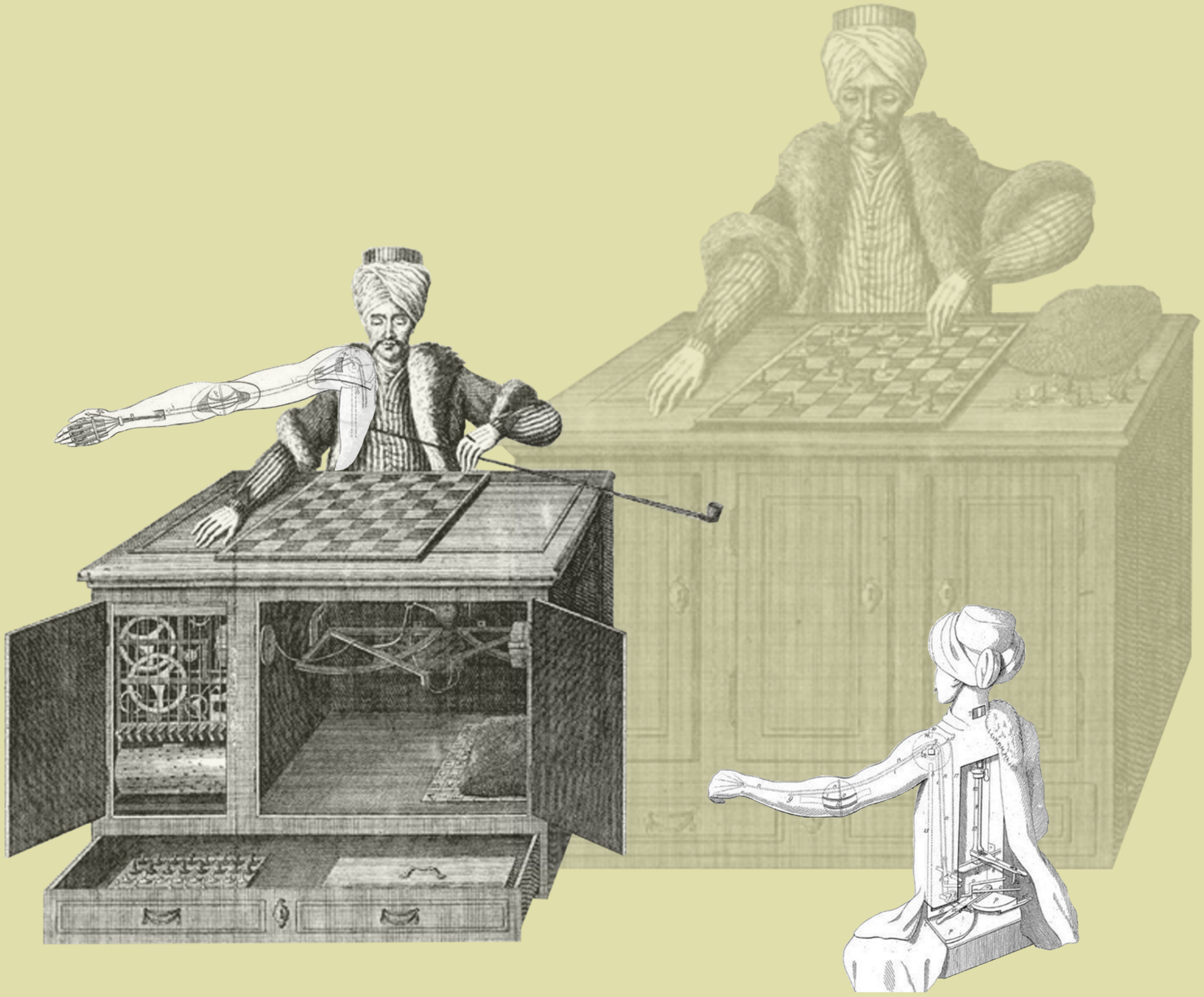
Solutions to App Development Challenges

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 2016-04-01

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

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:
>
> f0784a862295c031ccd1dfaee6d3e58201418153[6]

This will allow you to follow along and see exactly how the codebase 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/f0784a862295c031ccd1dfaee6d3e58201418153

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

Setting up

This book uses Marionette 2.3.2. 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 json
```

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:
>
> f0784a862295c031ccd1dfaee6d3e58201418153[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/f0784a862295c031ccd1dfaee6d3e58201418153.zip
[21]https://github.com/davidsulc/marionette-serious-progression-app/commit/f0784a862295c031ccd1dfaee6d3e58201418153

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)

```
1 ContactManager.on("before:start", function(){
2   _.templateSettings = {
3     interpolate: /\{\{=(.+?)\}\}/g,
4     escape: /\{\{-(.+?)\}\}/g,
5     evaluate: /\{\{(.+?)\}\}/g
6   };
7
8   var RegionContainer = Marionette.LayoutView.extend({
9     // edited for brevity
```

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 `<%...%>`

As you can tell, we've added this code to the "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.

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

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).

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

Updating the Underscore delimiters in index.html

```
<!-- <script type="text/template" id="header-template"> -->
<!--   ... -->
<!-- </script> -->

<script type="text/template" id="header-link">
  <a href="#{{- url }}">{{- name }}</a>
</script>

<!-- <script type="text/template" id="contact-list"> -->
<!--   ... -->
<!-- </script> -->

<script type="text/template" id="contact-list-none">
  <td colspan="3">No contacts to display.</td>
</script>

<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">
```

[24]http://underscorejs.org/#template
[25]http://en.wikipedia.org/wiki/Cross-site_scripting

```
      <i class="icon-pencil"></i>
      Edit
    </a>
    <button class="btn btn-small js-delete">
      <i class="icon-remove"></i>
      Delete
    </button>
  </td>
</script>

<script type="text/template" id="missing-contact-view">
  <div class="alert alert-error">This contact doesn't exist !</div>
</script>

<script type="text/template" id="contact-view">
  <h1>{{- firstName }} {{- lastName }}</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>
</script>

<script type="text/template" id="loading-view">
  <h1>{{- title }}</h1>
  <p>{{- message }}</p>
  <div id="spinner"></div>
</script>

<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:</label>
      <input id="contact-phoneNumber" name="phoneNumber"
                                         type="text" value="{{- phoneNumber }}"/>
    </div>
    <button class="btn js-submit">Save</button>
  </form>
</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("ContactManager.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("ContactManager.Entities.ContactCollection");
```

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

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

index.html

```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/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

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

var initializeContacts = function(){
  var contacts = new Entities.ContactCollection([
    { id:1, firstName: "Alice", lastName: "Arten", phoneNumber: "555-0184" },
    { id:2, firstName: "Bob", lastName: "Brigham", phoneNumber: "555-0163" },
    { id:3, firstName: "Charlie", lastName: "Campbell", phoneNumber: "555-0129" }
  ]);
  contacts.forEach(function(contact){
    contact.save();
  });
  return contacts.models;
};
```

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:

assets/js/entities/contact.js

```
// delete this function
~~var initializeContacts = function(){~~
~~  var 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){~~
```

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

```
    ~~contact.save();~~
  ~~});~~
  ~~return contacts.models;~~
~~};~~
```

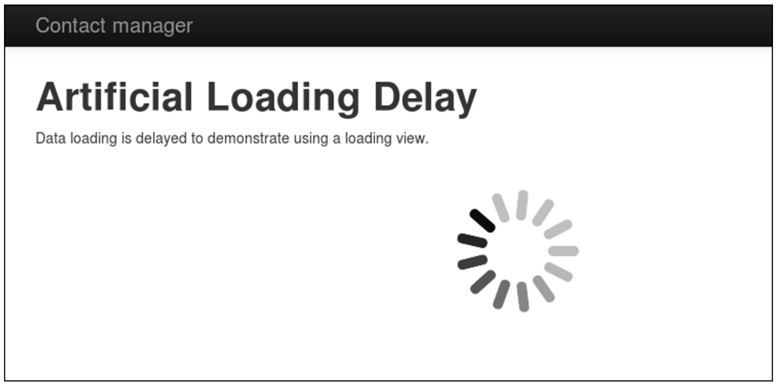
Since we no longer have an `initializeContacts` function, we need to adapt the rest of our code:

assets/js/entities/contact.js

```
 1 var API = {
 2   getContactEntities: function(){
 3     var contacts = new Entities.ContactCollection();
 4     var defer = $.Deferred();
 5     contacts.fetch({
 6       success: function(data){
 7         defer.resolve(data);
 8       }
 9     });
10     // delete these lines
11     ~~var promise = defer.promise();~~
12     ~~$.when(promise).done(function(fetchedContacts){~~
13       ~~if(fetchedContacts.length === 0){~~
14         ~~// if we don't have any contacts yet, create some for convenience~~
15         ~~var models = initializeContacts();~~
16         ~~contacts.reset(models);~~
17       ~~}~~
18     ~~});~~
19     ~~return promise;~~
20 
21     // return the promise
22     return defer.promise();
23   },
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.



Artificial Loading Delay

Data loading is delayed to demonstrate using a loading view.

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.regions.main.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.regions.main.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

```
 1  view.on("form:submit", function(data){
 2    if(contacts.length > 0){
 3      var highestId = contacts.max(function(c){ return c.id; }).get("id");
 4      data.id = highestId + 1;
 5    }
 6    else{
 7      data.id = 1;
 8    }
 9    if(newContact.save(data)){
10      contacts.add(newContact);
11      // 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:

0affef7f4bb575d4e8fd640e2e9cc9e176bc5079[28]

[28]https://github.com/davidsulc/marionette-serious-progression-app/commit/0affef7f4bb575d4e8fd640e2e9cc9e176bc5079

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;
    }
  },

  // 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

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

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

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

```
<script type="text/template" id="contact-view">
  <h1>{{- data.fullName }}</h1>

  <!-- edited for brevity -->
</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

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

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

  // edited for brevity
});
```

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:</label>
      <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:

8610f08d2d15dd3dd91fd5efdb621969fb3e13e9[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/8610f08d2d15dd3dd91fd5efdb621969fb3e13e9

```
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.

Chapters not in Sample

This is a sample of the book, several chapters are absent.

You can get the complete book at https://leanpub.com/marionette-gentle-introduction/[31].

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