iOS *on* Rails

The reference for writing superb iOS apps with Ruby on Rails backends.

iOS on Rails (Beta)

April 16, 2014

Contents

About this book

The book is currently in Beta. These means we are actively writing and revising content and the example applications. You'll receive all updates to the book and have direct access to the GitHub repository for the book and example applications, where you can watch the magic happen and file GitHub issues.

Welcome to the iOS on Rails eBook sample. This is published directly from the book, so that you can get a sense for the content, style, and delivery of the product. We've included three sample sections. One is specific to Rails and shows how to handle GET requests. The last two are iOS specific, and cover creating your API client from scratch or with AFNetworking.

If you enjoy the sample, you can get access to the entire book and sample application at:

`https://learn.thoughtbot.com/products/25-ios-on-rails`

The eBook covers intermediate to advanced topics on creating iOS client and Ruby on Rails server applications.

In addition to the book (in HTML, PDF, EPUB, and Kindle formats), you also get two complete example applications, an iOS and a Rails app.

The book is written using Markdown and pandoc, and hosted on GitHub. You get access to all this. You can also use the GitHub comment and issue features to give us feedback about what we've written and what you'd like to see.

Contact us

If you have any questions, or just want to get in touch, drop us a line at learn@thoughtbot.com.

Introduction

Why this book?

There are many ways to build the backend for an iOS application but you only need one. And depending on the complexity of the API you are going to create, different solutions work best for different applications.

Just as Rails makes it possible to set up a basic web application in a matter of minutes, Rails makes it possible to set up a basic API in a matter of minutes. But deciding how to structure your API isn't easy. While experimenting with all of the options is a fun weekend project, sometimes you just want to get going. This book will help you do just that. While your API will no doubt require some tweaking while fleshing out your iOS app, the approach we will be taking is to define and build the API first, and then consume this API through our iOS app.

The Rails portions of iOS on Rails will guide you through what we have found to be a robust, clean, flexible way of building out a JSON API with Rails. We provide code samples for GET, POST, and PATCH requests. In addition, we will explore some of the alternative approaches that we didn't choose and explain why we made the choices that we did.

The iOS portion of the book will then walk, step-by-step, through creating an iOS application that works with the Rails API you just created. The iOS application will use each endpoint to post up objects and get back necessary data for the user. Our model objects in the iOS app will correspond with the model objects in the database, and be populated with response data from the API.

Who is this book for?

This book is for a developer who wants to build an iOS application with a Rails backend. It's also a book for both a Rails developer and an iOS developer to share and use in concert to create an app quickly and with more flexibility to change it than a backend-as-a-service provider like StackMob or Parse.

The approach shared in this book is the result of our own experiments as Rails and iOS developers working together to build an application. The Rails portions of this book assume a basic working knowledge of how to build a web application with Rails as well as the Ruby programming language. The iOS portions of this book assume experience with object oriented programming and a basic familiarity with the Objective-C programming language.

Part I

Building the Humon Rails App

Creating a GET request

It all starts with a request spec

At thoughtbot, we do test-driven and Outside-in development, which means that we start work on any feature by writing a high-level test that describes user behaviors. You can read a more detailed description of Outside-in development here, but the benefits can be summarized as follows:

> Outside-in along with the test-driven process helps you write just the minimum amount of code that provides value to stakeholders, and not a line more.

The external interface of our application will be the iOS app that GETs and POSTs data to the Rails app, so feature specs, which usually interact with the application via web interfaces, do not make sense. Jonas Nicklas, the creator of Capybara, said it best: "Do not test APIs with Capybara. It wasn't designed for it."

Instead, we will use request specs. RSpec request specs, like feature specs, are a great way to ensure the entire stack is working together properly but via HTTP verbs, response codes, and responses rather than browser interactions.

When writing our request specs, we found that we were calling `JSON.parse(response.body)` over and over again. We abstracted this into a method called `response_json`, which we use below and in all of our request specs that include a JSON response.

```
# spec/requests/api/v1/events/events_spec.rb

require 'spec_helper'

describe 'GET /v1/events/:id' do
  it 'returns an event by :id' do
    event = create(:event)

    get "/v1/events/#{event.id}"
```

```
    expect(response_json).to eq(
      {
        'address' => event.address,
        'ended_at' => event.ended_at,
        'id' => event.id,
        'lat' => event.lat,
        'lon' => event.lon,
        'name' => event.name,
        'started_at' => event.started_at.as_json,
        'owner' => {
          'device_token' => event.owner.device_token
        }
      }
    )
  end
end
```

Model

This first error we will get for the request spec above is that our app does not have a factory named `event`. FactoryGirl guesses the class of the object based on the factory name, so creating the `event` factory is a good opportunity to set up our `Event` model.

At the model level, Rails applications that serve a JSON API look exactly like regular web applications built with Rails. Although the views and controllers will be versioned, we will write our migrations like standard Rails migrations and keep our models within the models directory. You can see the data migrations for our example application here.

At this point, let's assume our `User` model has already been created.

Our `Event` model has a few validations and relations, so we will write tests for those validations. In our development process, we would write the following tests line-by-line, watching them fail, and writing the lines in our model one at a time to make them pass. We will use FactoryGirl, Shoulda Matchers, and RSpec for our unit tests. To see our full test setup, see our `spec_helper` here.

```
# spec/models/event.rb
```

```
require 'spec_helper'

describe Event, 'Validations' do
  it { should validate_presence_of(:lat) }
  it { should validate_presence_of(:lon) }
  it { should validate_presence_of(:name) }
  it { should validate_presence_of(:started_at) }
end

describe Event, 'Associations' do
  it { should have_many(:attendances) }
  it { should belong_to(:owner).class_name('User') }
end
```

To make the tests pass, we will write a migration (note: your file name will be different, as the numbers in the name are generated based on the date and time the migration was created):

```
# db/migrate/20131028210819_create_events.rb

class CreateEvents < ActiveRecord::Migration
  def change
    create_table :events do |t|
      t.timestamps null: false
      t.string :address
      t.datetime :ended_at
      t.float :lat, null: false
      t.float :lon, null: false
      t.string :name, null: false
      t.datetime :started_at, null: false
      t.integer :user_id, null: false
    end

    add_index :events, :user_id
  end
end
```

and add those validations to the model:

```
# app/models/event.rb

class Event < ActiveRecord::Base
  validates :lat, presence: true
  validates :lon, presence: true
  validates :name, presence: true
  validates :started_at, presence: true

  belongs_to :owner, foreign_key: 'user_id', class_name: 'User'
end
```

Once this is working, we can add the `event` Factory to `spec/factories.rb` for use in our request spec.

Controller

At this point, we can create an `event` object using FactoryGirl, but our request spec is failing on the next line. This is because we have no routes set up for the path we are using in our test's GET request (`get "/v1/events/#{event.id}"`). To fix this, we need to add a controller and configure our `routes.rb` file.

As we discussed in the versioning section of our introduction, we will add controllers within `api/v1` directory so that we may release future versions of our API without breaking older versions of our application.

Because our `routes.rb` file tells our controllers to look for the JSON format by default, we do not need to tell our individual controllers to render JSON templates. We do, however, need to add our new paths to our routes file:

```
# config/routes.rb

Humon::Application.routes.draw do
  scope module: :api, defaults: { format: 'json' } do
    namespace :v1 do
      resources :events, only: [:show]
```

```
    end
  end
end
```

Aside from including our controller within the `api/v1 directory`, our `EventsController` looks much like a standard Rails controller. To make our request spec pass, we need to add a single action to our API:

```
# app/controllers/api/v1/events_controller.rb

class Api::V1::EventsController < ApplicationController
  def show
    @event = Event.find(params[:id])
  end
end
```

View

Our controller and routes are set up, but we still need one final piece before our spec will pass: a view. Our request spec is looking for a view template with some response JSON, but so we need to create that view.

For a Rails developer, the views are where there will be the most difference between a standard web application and a JSON API. As with our controllers, we will include our views in the `api/v1` directory so that they are versioned.

Just like regular view partials, Jbuilder partials minimize duplication by letting us re-use blocks of view code in many different places. JSON representations of data frequently include duplication (a collection is usually an array of the same JSON structure that would be found for a single object), so partials are especially handy when creating a JSON API. We will use Jbuilder's DSL to tell our show view to find the event partial:

```
# app/views/api/v1/events/show.json.jbuilder

json.partial! 'event', event: @event
```

Our show GET view is looking for a partial named `_event.json.jbuilder` within the `events` directory. So we will create that partial next:

```
# app/views/api/v1/events/_event.json.jbuilder

json.cache! event do
  json.address event.address
  json.ended_at event.ended_at
  json.id event.id
  json.lat event.lat
  json.lon event.lon
  json.name event.name
  json.started_at event.started_at

  json.owner do
    json.device_token event.owner.device_token
  end
end
```

Caching our view

You might be wondering what the `json.cache!` at the top of our `event` partial is doing. Jbuilder supports fragment caching, and you tell your app to cache a block of view code by wrapping it in a `json.cache!` block. While the load time for the JSON in our view above is going to be teeny tiny, adding fragment caching is simple and a good habit to get into for apps that are likely to expand over time.

If you're interested in learning more about fragment caching, there is a great Railscast (paid) on the topic.

Putting it all together

We have now successfully created our first API endpoint for Humon and our request spec should pass!

But let's test it manually just to make sure. Our iOS app isn't up and running yet, so we will have to create records in Rails console. Make sure you are in your project directory in Terminal, run `rails console` and the enter the following:

```
User.create(device_token: '12345')
Event.create(
  address: '85 2nd Street',
  lat: 37.8050217,
  lon: -122.409155,
  name: 'Best event OF ALL TIME!',
  owner: User.find_by(device_token: '12345'),
  started_at: Time.zone.now
)
```

Assuming this created your first event (`id` will equal 1) and you are running `rails server` in Terminal (you will need to exit from Rails console or open a new Terminal window to do this), when you visit `localhost:3000/v1/events/1` in your browser you should see something like this:

```
{
  "address":"85 2nd Street",
  "ended_at":"2013-09-17T00:00:00.000Z",
  "id":1,
  "lat":37.8050217,
  "lon":-122.409155,
  "name":"Best event OF ALL TIME!",
  "started_at":"2013-09-16T00:00:00.000Z",
  "owner":{
    "device_token":"234324235"
   }
 }
```

Alternatively, you can run a curl request (`curl http://localhost:3000/v1/events/1`) from Terminal and see the same JSON output.

Congratulations, you just created your first API endpoint with Rails!

Part II

Building the Humon iOS App

A Rails API Client With NSURLSession

Before we go about making our first API request, we need to decide how we are going to make our networking calls. As mentioned in the Cocoapods chapter, the AFNetworking framework is a clean and reliable solution to making networking requests. We will be using AFNetworking in this book, but we'll also include examples of how to make a few API requests manually for reference. AFNetworking brings a lot more to the table than just wrapping up your network requests; but, like a programming planeteer, the choice is yours.

Creating a Singleton Client Object

Create a subclass of NSObject called HUMRailsClient. All of our API requests will be handled by one instance of the HUMRailsClient, so we're going to create a singleton of HUMRailsClient called sharedClient. What we will create and refer to as a singleton isn't a dictionary-definition singleton, since we aren't completely limiting the instantiation of HUMRailsClient to only one object. We are, however, limiting the instantiation of HUMRailsClient to only one object if we always use our sharedClient. Essentially, our sharedClient is a singleton if we use it consistantly but is not if we errantly decide to instantiate another instance of HUMRailsClient using [[HUMRailsClient alloc] init].

Declare a class method that will return our singleton by adding `+ (instancetype)sharedClient;` to your HUMRailsClient.h file. We use instancetype as our return type to indicate that this class method will return an instance of HUMRailsClient. The + indicates that sharedClient is a class method to be called directly on the HUMRailsClient class. Prepending your class method with "shared" indicates to other developers that the method returns a singleton.

Now let's implement this method:

```
// HUMRailsClient.m

+ (instancetype)sharedClient
{
    static HUMRailsClient *_sharedClient = nil;
```

```
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{

        // Code to be run only once
        _sharedClient = [[HUMRailsClient alloc] init];

    });

    return _sharedClient;
}
```

First, we declare a static variable of type HUMRailsClient. Since it's a static variable, _sharedClient will last for the life of the program.

Then, we use Grand Central Dispatch to execute a block of code once and only once. If you are using XCode and begin typing dispatch_once, you can even use autocomplete to find and insert the entire dispatch_once code snippet. dispatch_once takes a reference to a static variable of type dispatch_once_t and a block of code to execute. dispatch_once_t is a long variable type that indicates whether or not the block of code has been executed already. On the first call of dispatch_once, the onceToken is set and the block executed, but on every subsequent call the block is not executed because the onceToken has already been set.

Inside the block we instantiate a HUMRailsClient and set it as the value of the static variable sharedClient. Once that is done, we simply need to return our singleton sharedClient.

Creating a Session for Handling Requests

iOS7 introduced the NSURLSessions class, which is an object that handles groups of HTTP requests. Each API request we make in a NSURLSession is encapsulated in a NSURLSessionTask, which executes the request asynchronously and notifies you of completion by executing a block or by calling a method on its delegate.

There are three different types of NSURLSessions, including one that allows your app to continue downloading data even if your app is in the background.

The type of a session is determined by its `sessionConfiguration`, but for simple API requests we only need to use the default session type.

Declare a session property and a static app secret string by placing:

```
// HUMRailsClient.m

static NSString *const HUMAppSecret =
    @"yourOwnUniqueAppSecretThatYouShouldRandomlyGenerateAndKeepSecret";

@interface HUMRailsClient ()

@property (strong, nonatomic) NSURLSession *session;

@end
```

above your @implementation inside of HUMRailsClient.m. We will use the `HUMAppSecret` to sign POST requests to /users so that the backend can validate that the request is coming from the mobile app.

Then, overwrite the HUMRailsClient's init method to set the client's `session` when we initialize it:

```
// HUMRailsClient.m

- (instancetype)init
{
    self = [super init];

    if (!self)
        return nil;

    // Create a session configuration
    NSURLSessionConfiguration *sessionConfiguration =
        [NSURLSessionConfiguration defaultSessionConfiguration];
    sessionConfiguration.timeoutIntervalForRequest = 30.0;
    sessionConfiguration.timeoutIntervalForResource = 30.0;
```

```
    //Set the session headers
    NSDictionary *headers = [HUMUserSession userID] ?
        @{
          @"Accept" : @"application/json",
          @"Content-Type" : @"application/json",
          @"X-DEVICE-TOKEN" : [HUMUserSession userID]
          } :
        @{
          @"Accept" : @"application/json",
          @"Content-Type" : @"application/json",
          @"X-APP-SECRET" : HUMAppSecret
          };
    [sessionConfiguration setHTTPAdditionalHeaders:headers];

    // Create a session
    _session = [NSURLSession sessionWithConfiguration:sessionConfiguration];

    return self;
}
```

This custom init method creates a `sessionConfiguration`, uses the `sessionConfiguration` to create an NSURLSession, and sets the session on the `_sharedClient`.

We could just use the default NSURLSessionConfiguration that is returned from NSURLSessionConfiguration's class method `defaultSessionConfiguration` to create our NSURLSession. However, we also want to change our timeout properties to 30 seconds and add some HTTP headers that we will be sending and receiving JSON from our API.

Setting the Session Headers

Setting the session headers on the `sessionConfiguration` is particularly important, since sending the app secret is necessary for user creation, while the user's ID is necessary for all other requests. When we initialize the `sharedClient` singleton, we place the user ID in the header if we've already saved one in the keychain, or the app secret if there is no user ID saved in the keychain. Having

the app secret in the header is only necessary for the POST to /users request, so we'll change out the app secret header one we have successfully made that request

A Rails API Client With AFNetworking

Now that we've created our own networking client, let's see how we could do this using the AFNetworking framework. We'll create another client that is a subclass of AFNetworking's session manager instead of NSObject.

Creating a Singleton Client Object

Create a subclass of AFHTTPSessionManager called HUMRailsAFNClient. Declare a class method that will return a shared client singleton as we did in our other client by adding `+ (instancetype)sharedClient;` to your HUMRailsAFNClient.h file. The implementation of this method looks similar as well:

```
// HUMRailsAFNClient.m

+ (instancetype)sharedClient
{
    static HUMRailsAFNClient *_sharedClient = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{

        // Create a client
        NSURL *baseURL = [NSURL URLWithString:ROOT_URL];
        _sharedClient = [[HUMRailsAFNClient alloc] initWithBaseURL:baseURL];

        // Set the client header fields
        if ([HUMUserSession userID])
           [_sharedClient.requestSerializer setValue:[HUMUserSession userID]
                                  forHTTPHeaderField:@"X-DEVICE-TOKEN"];
        else
            [_sharedClient.requestSerializer setValue:HUMAppSecret
```

```
                                forHTTPHeaderField:@"X-APP-SECRET"];

    });

    return _sharedClient;
}
```

With AFNetworking, we don't have to manually set up the session configuration and session with our own custom init method. We simply initialize the client using `initWithBaseURL:`, which means that our paths later will be relative to this ROOT_URL.

Setting the Session Headers

As before, we need to set the user's ID in the header if we have already created a user for this device. If not, we set the app secret so that we can make a POST to /users to create a user with the app secret.

Closing

Thanks for checking out the sample of our iOS on Rails eBook. If you'd like to get access to the full content, the example applications, ongoing updates, you can pick it up on our website:

https://learn.thoughtbot.com/products/25-ios-on-rails