Lab 2.1: REST/JSON APIs for Data Models
IN705 Databases Three

Introduction

Last week we built a simple “Hello, world" application with Rails. This application didn’t even need to use a data model. Now we will use Rails to build a REST/JSON API around a basic data model.

1 Install the rails-api gem

Since we are building a REST/JSON web service instead of a full web site, we don’t need all of the features of a typical rails application. In particular, we won’t use views. To facilitate building this application, we’ll intall the `rails-api` gem on our EC2 servers with the command

```
sudo gem install rails-api
```

2 Creating the new application

Our application will be a knockoff of Twitter, so we’ll give it a similar name and call it *Splatter*. Inside your db3 directory (which is now tracked in a GitHub repository, right?), create the new application with the command

```
rails-api new splatter
```

Notice that we use the new `rails-api` command rather than the typical `rails` one.

3 Creating our first model

You’re probably familiar with Twitter and know that, basically, it has a bunch of users who post short messages. Our application will work in the same way, so we’ll start by creating a model for users.

First, a disclaimer: If you are working with almost any web development framework and you find yourself creating your own user class from scratch, you’re probably doing it wrong. Frameworks usually have very good built-in user classes, or they have plugins that provide them. For example, *Devise* is an execellent plugin for handling users in Rails.

We will build our own user class for two reasons: First, because it will help us to learn a few things that would be harder to see if we used a tool like Devise. Second, because from the API server point of view, our user objects will behave a little differently than a typical web application user.

Our user model will have four fields to start with:

1. email: string
2. password[1]: string
3. name: string, used for display purposes
4. blurb: text, a description-like field for the user profile

We could write this model class and the associated controller class by hand, but we'll have Rails generate the initial versions for us with the following command (Run this, and the commands that follow, from within your `splatter` directory.)

```
rails generate scaffold User email:string password:string name:string blurb:text
```

This generates several things for us. In fact, we are only two commands away from running a fully functional REST/API service! The three we are interested in for now are the model class, the controller class, and the database mirgration.

3.1 The model

Inspect the model class in `app/models/user.rb`. There's astonishly little going on there. Our `User` class inherits from the `ActiveRecord::Base` class. For the time being we don't need to override or extend fromthe base class in any way.

3.2 The migration

The database migration file is in `db/migrations/<timestamp>_create_users.rb`. The value of the timestamp portion depends on the time at which we generated the migration. This file contains code to modify the database to suit our model. Our current migration will create a new table with the fields we specified earlier. It also gives the model some standard timestamps.

To apply the migration to the database, run the command

```
rake db:migrate
```

Right now we're using sqlite for early development. Later we'll see that we can switch to a postgres database by modifying our config file and running the migrations.

3.3 The controller

Examine the controller class in `app/controllers/users_controller.rb`. We see that the controller contains methods to create, read, update, and delete user records, as well as an index method that returns a list of all the users. The comments also show the HTTP methods and URLs used to invoke the controller actions.

The API service is nearly ready to run, but we need to add something to the controller. The controller will not pass any parameters to the the User class' `new` or `update` methods unless those parameters are whitelisted. To do this, add the following code at the bottom of the class, just before the final `end`.

```
private

  def user_params(params)
    params.permit(:email, :password, :name, :blurb)
  end
```

[1]Storing passwords in plain text is a Bad Thing. We'll leave ours in plain text for now and deal with that later.

Anywhere in the controller where you see `params[:user]` passed into calls to `new` or `update`, change the argument to `user_params(params[:user])`.

4 Try out your application using curl

To try out our application we need to send it various HTTP requests, but it's not easy to generate those requests usign a tool like a web browser. There are some useful browser plugins, but for our purposes it's easier to just use `curl`, a command line utility for making HTTP requests. Curl is installed on your EC2 servers, so you can try it in an ssh session.

Start your server with the `rails server` command like we did in the previous lab. In a second ssh session, enter the following:

```
curl -i -H "Content-type: application/json" -X POST http://lastname.sqrawler.com:3000/users -d
'{"user": {"email":"test@foo.com", "name":"Test User", "password":"foo"}}'
```

substituting your own last name in the URL.

Note the value of `id` in the returned JSON.

Now view your new user with the command

```
curl -i -H "Content-type: application/json" -X GET http://lastname.sqrawler.com:3000/users/1
```

and then delete it with the command

```
curl -i -H "Content-type: application/json" -X DELETE http://lastname.sqrawler.com:3000/users/1
```

where you may need to substitute the number 1 above with the id of your user.

What curl command would you use to update an exisiting user? Experiment with your server and the curl command to get a sense for how things work.

5 Wrapping up

Our web service is still very crude and it will take a few iterations until we have a satisfactory version for real use. Don't forget to commit your new code to your GitHub repository.