

Getting Started with Ruby on Rails

Connecting to the server

(Substitute your login name and the appropriate development server domain name for "yourlogin" and "yourdevserver.com".)

Mac OS

Using the "Terminal" application:

```
ssh yourlogin@yourdevserver.com
```

Example:

```
ssh jkupp@waycoolprojects.com
```

Enter your password when prompted (you won't see anything as you type).

Windows

Download and install the Putty terminal application. Open and use the following settings:

Host name: *yourdevserver.com*

Port: 22

Connection type: SSH

Enter your login and password when prompted.

Getting around the file system

<code>cd path</code>	Change directory to <i>path</i> .
Examples:	
<code>cd jkupp</code>	Change to directory "jkupp", which is in the current directory.
<code>cd /rails/jkupp</code>	Change to the directory "jkupp" inside directory "rails"
<code>cd ..</code>	Change to one directory up in the file system.
<code>ls</code>	List the contents of the current directory.
<code>ls -l</code>	Same, but with details about each item.
<code>mkdir directoryname</code>	Make a directory with a name of your choosing.
<code>rm filename</code>	Remove a file (Use with caution: there is no confirmation or undo!)
<code>rm -R directoryname</code>	Remove a directory and everything inside it.
<code>mv oldpath newpath</code>	Move or rename a file or directory at <i>oldpath</i> to <i>newpath</i>

Starting a new Rails application

1. Using Terminal or Putty, ssh into the server and cd into your rails directory. For example (substitute your user name for “jkupp”):

```
ssh jkupp@rorty.umflint.edu
(enter password)
cd /rails/jkupp
```

2. Enter the following command:

```
rails new app_name -d mysql
```

For example, for my “cabinet of curiosity” site, I could enter

```
rails new curiosity -d mysql
```

Rails will make a new directory called “curiosity,” and inside it a skeleton set of files and sub-directories. The “-d mysql” parameter sets things up to connect to a MySQL database.

3. Now, cd into the root of the new directory. This is where you will need to be in order for the rest of the commands to work.

```
cd app_name (e.g.: cd curiosity)
```

4. We’ll need to leave Terminal/Putty for a moment to edit the database connector file. Using Komodo Edit, connect to the server and navigate to the directory you just made. Now go into the “config” directory and open the file called “database.yml”. In the “development” section, change the “database,” “username,” and “password” lines. You will have a database already set up and named the same as your username, with access granted to your username and password. So in my case, it would look like this:

```
development:
  adapter: mysql2
  encoding: utf8
  reconnect: false
  database: jkupp
  pool: 5
  username: jkupp
  password: (our usual password for this class)
  socket: /tmp/mysql.sock
```

Caution: be sure to keep the space after the colon on each line. Save the file.

5. Now let's go back to the terminal. Start your Rails app with the following command:

```
rails server -p port_number
```

For *port_number*, substitute any number between 4000 and 4999. If someone else is using the same number, you'll get a message saying that the port is in use, and the app won't start.

Once your app is started, you should be able to call it up in a web browser. For example, if my app is running on port 4001, I would use the following URL:

```
http://rorty.umflint.edu:4001
```

You should see a "Welcome aboard" message.

Go back to your terminal and enter `ctrl-C` to stop the app.

6. Let's make our first data object. My Cabinet of Curiosity will have a lot of items in it, so let's think about the information an "item" should have:

author	This is going to be short bit of text, or "string," containing the name of the author.
title	Another string, containing the title.
description	A long bit of text, describing the item.

In the terminal, I'll put in this command:

```
rails generate scaffold Item author:string title:string description:text
```

That creates a model, a controller with some default actions, and a set of default views for the "Item" object class. It also makes a "migration" -- a set of instructions for making or modifying a table in the database. We still need to "migrate" those instructions over to the database, though, and to do so, we use the "rake" command:

```
rake db:migrate
```

Start up the app again (using the "rails server" command from step 5), switch over to your browser, and this time go to:

```
http://rorty.umflint.edu:4001/items
```

You now have a working (though not beautiful) database-driven website. Congrats! Remember to stop the application with `ctrl-C` when you are done.

(By the way, you may be wondering how you will go from a website you have to start and stop using a command line, with an ugly port number at the end of the domain name, to something more permanent and easier to get to. Rest assured, eventually you will be able to move your site to a "production server," where it will get its own domain or sub-domain, and be set up to be always running.)

Appendix: A tour of your Rails application directories:

app	The "heart" of your application code, including models, controllers, views, and helper methods.
controllers	Files that contain methods ("actions") -- the central logic of your app
helpers	"Auxiliary" methods that aren't part of the main code.
mailers	Code for sending email from your app.
models	A file for each type of data object.
views	Files that take care of the visual layout you see in your browser. In general, there is one view file for each action. Contains sub-directories corresponding to each controller, plus a "layouts" directory for the static parts of your interface.
config	Files that contain settings and preferences, including database.yml, which defines connections to a database.
db	Contains a "migrate" directory, which in turn contains migration files -- instructions for creating and modifying tables in the database.
doc, lib	You'll never use these.
log	Contains log files for your app
public	Static items that can be accessed directly from the web. Images, javascripts, CSS files, and other static files go here.
script	You will not need to do anything in this directory.
test	Used for testing functions.
tmp	For system use.
vendor	Place to put plugins, etc.

Part II: Item meets Comment: a serious relationship

1. Now that we have Items, it would be nice to be able to add comments. Eventually we'll want a way for users to sign in to our site, but we'll worry about that later. Rails makes it easy to get pieces of a site up and running quickly, so we can change them later without losing too much work.

We'll want a separate model (type of data object) for the comments, and let's have it contain the following information:

author	A string of characters.
content	The content of the comment -- a bit of text.
item_id	An integer that indicates what item the comment belongs to.

We'll use the "scaffold" command again:

```
rails generate scaffold Comment author:string content:string
item_id:integer
```

Then:

```
rake db:migrate
```

... to make a table for comments in the database.

2. Now things get a little tricky. First, we have to tell the Item model and Comment model about each other. We do the introduction in the files `comment.rb` and `item.rb`, in the `models` directory. Add a line in the middle of each of these files, so they look like this:

`comment.rb`:

```
class Comment < ActiveRecord::Base
  belongs_to :item
end
```

`item.rb`:

```
class Item < ActiveRecord::Base
  has_many :comments
end
```

Notice that a Comment "belongs_to" (an) item (singular), while an Item "has_many" comments (plural). Also note that we don't need a "comment_id" in the Item model -- the `item_id` in the Comment model is enough to be able to know who goes with whom.

3. Now we need a form so users can add comments. The scaffolding includes one (which you can see at [http://\[server domain and port\]/comments/new](http://[server domain and port]/comments/new)), but we don't want the user to have to type in the item ID, and anyway we want to have the form show up underneath the appropriate item.

We can use a lot of the form code from the scaffold, though -- let's copy all the code from `/app/views/comments/_form.html.erb` and, with a few changes, put it at the bottom of `/app/views/items/show.html.erb`

```
<%= form_for(@comment) do |f| %>
  <% if @comment.errors.any? %>
    <div id="error_explanation">
      <h2><%= pluralize(@comment.errors.count, "error") %>
        prohibited this comment from being saved:</h2>

      <ul>
        <% @comment.errors.full_messages.each do |msg| %>
          <li><%= msg %></li>
        <% end %>
      </ul>
    </div>
  <% end %>

  <h3>Add a comment:</h3>

  <div class="field">
    <%= f.label "Your name" %><br />
    <%= f.text_field :author %>
  </div>

  <div class="field">
    <%= f.label "Comment" %><br />
    <%= f.text_area :content %>
  </div>

  <%= f.hidden_field :item_id, :value=>@item.id %>

  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>
```

I added or changed the parts in bold, most notably changing the "item_id" field to a hidden field (one that doesn't show up to the user), containing the ID of the item. We want to submit that information so it can be put into the item_id field of the comment we are creating.

4. We need to do one more thing before all this will work. Notice at the top of the form, the `form_for` method expects an instance variable, `@comment`. (Form_for does a lot of the work of creating the HTML form for us, based on the data object we specify.) We need to set that in the action that calls up this view, in this case the "show" action in the `items_controller`. We do that by adding the line

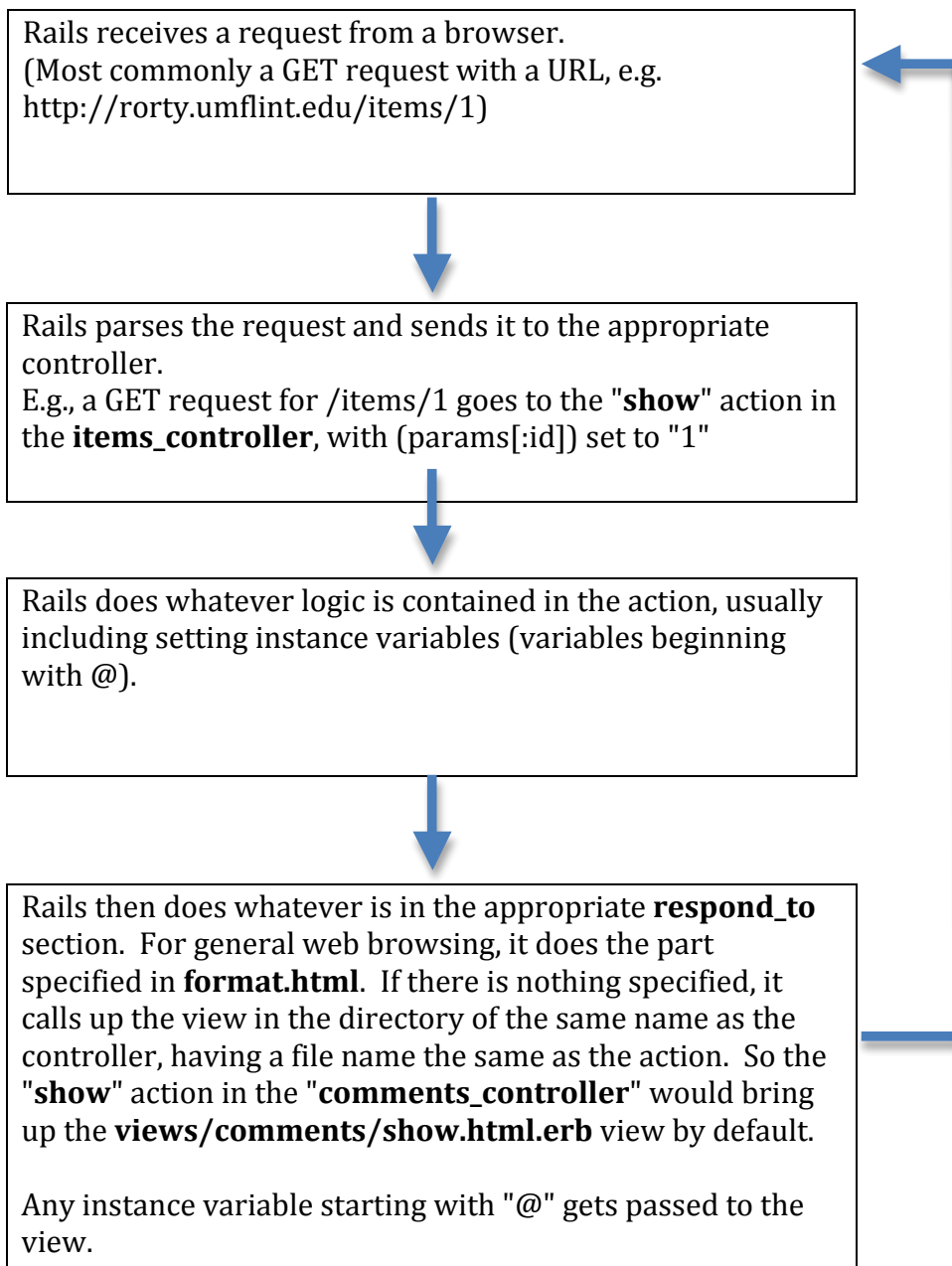
```
@comment = Comment.new
```

...which makes a new, "empty" instance of Comment, and puts it in the @comment variable.

items_controller.rb

```
.  
.br/>.br/>  
# GET /items/1  
# GET /items/1.xml  
def show  
  @item = Item.find(params[:id])  
  @comment = Comment.new  
  respond_to do |format|  
    format.html # show.html.erb  
    format.xml { render :xml => @item }  
  end  
end  
  
.br/>.br/>
```

Let's pause a moment and look at how the controllers and views work together.



Now one can add a comment, using the form on in the views/items/show.html.erb view that we edited earlier. Because we generated the form with **form_for(@comment)**, when we submit the form, the request by default invokes the "create" action in the **comments_controller**.

```
# POST /comments
# POST /comments.xml
def create
  @comment = Comment.new(params[:comment])

  respond_to do |format|
    if @comment.save
      format.html { redirect_to(@comment, :notice => 'Comment
was successfully created.') }
      format.xml { render :xml => @comment, :status =>
:created, :location => @comment }
    else
      format.html { render :action => "new" }
      format.xml { render :xml => @comment.errors, :status =>
:unprocessable_entity }
    end
  end
end
end
```

The problem is, when the comment is successfully saved, the respond_to method says to redirect to "@comment," which by default brings up the "show" action in the comments_controller. But what we really want is to show the *item* to which the comment belongs.

Fortunately, we've told the Comment model that comments can belong to items. So instead of invoking @comment we just need to invoke the item that goes with it. In Rails, it turns out, that's easy. We can simply write "**@comment.item**".

The format.html line then becomes:

```
format.html { redirect_to(@comment.item, :notice => 'Comment was
successfully created.') }
```

While we're at it, let's put in a few snippets of code so that "notice" and error messages show up.

app/views/layouts/application.html.erb

```
...
<div id="content">
  <p class="notice"><%= notice %></p>
  <p class="alert"><%= alert %></p>
  <%= yield %>
</div>
...
```

4. Whew! That was a lot of work. But there's still one more thing we need to do. We want to show all the comments for a particular item in the view for that item. We've already seen how to loop through a collection of objects: in `views/items/index.html.erb` we had:

```
<% @items.each do |item| %>
.
.
.
<% end %>
```

Now, though, we want to do the same thing with the set of comments that belongs to a particular item. It turns out that is easy, too, because we've told the `Item` model that it can have many comments. So in the **`views/items/show.html.erb`** view, we can insert something like this:

```
<h3>Current comments:</h3>
<% @item.comments.each do |comment| %>
  <p><%= comment.author %>: <%= comment.content %></p>
<% end %>
```

Of course, that's a pretty crude way to display the comments, but we can pretty it up later. And there are lots of other things missing now that we would eventually want: for example, links to edit and delete comments, not to mention user registration and logins (right now anyone can pretend to be anyone). But it's a start!

Part III: Attaching a photo

File uploads used to be insanely difficult to code. Fortunately, now there are that make it one of the easier things in Rails. We are going to use a plugin called "Paperclip" (others are "attachment_fu" and "CarrierWave"). We're going to put a place for attaching photos right into the Item model. The limitation is that we'll only be able to attach one photo to any given item. (Later, we'll work on modifications so that we can attach more than one.)

I've already installed the Paperclip "gem" on our server, and to access it, you just need to put this line (anywhere) in the "Gemfile," which you will find near the bottom of your file list in Komodo Edit:

```
gem "paperclip", "~> 2.3"
```

(If you have your app running, you may need to stop and restart it for this line to take effect.)

Now add this code in the Item model (app/models/item.rb):

```
has_attached_file :photo,  
  :styles => {  
    :thumb=> "100x100#",  
    :small => "150x150"> } }
```

The "100x100#" will create an image format called ":thumb," 100 pix by 100px, cropping the height if necessary, and a format called ":small," which is 150 pix tall and no more than 150 pix wide, keeping the original proportions.

Now we need to add some columns to the items table in our database. We'll use "rails generate migration" at the command line:

```
rails generate migration AddPhotoToItems photo_file_name:string  
photo_content_type:string photo_file_size:integer
```

(That's all one line.)

Then still at the command line:

```
rake db:migrate
```

Now let's tweak the items form a bit (`app/views/items/_form.html`). Changed items are in bold.

```
<%= form_for(@item, :html => { :multipart => true }) do |f| %>
  <% if @item.errors.any? %>
    <div id="error_explanation">
      <h2><%= pluralize(@item.errors.count, "error") %>
        prohibited this item from being saved:</h2>
      <ul>
        <% @item.errors.full_messages.each do |msg| %>
          <li><%= msg %></li>
        <% end %>
      </ul>
    </div>
  <% end %>

  <div class="field">
    <%= f.label :title %><br />
    <%= f.text_field :title %>
  </div>
  <div class="field">
    <%= f.label :author %><br />
    <%= f.text_field :author %>
  </div>
  <div class="field">
    <%= f.label :description %><br />
    <%= f.text_area :description %>
  </div>
  <div class="field">
    <%= f.label "Attach a photo" %><br />
    <%= f.file_field :photo %>
  </div>
  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>
```

To show the photo, use code like this, for example somewhere in `app/views/items/show.html.erb`:

```
<% if @item.photo.exists? then %>
  <%= image_tag @item.photo.url(:small) %>
<% end %>
```

To show the thumbnail version instead, replace `:small` with `:thumb`.

That's it? Really? Yep.

Part IV: Users

It's fairly certain that whatever website you build, you will want some way for users to log in and maintain an identity on your site. Even if your site is mostly informational, you'll probably want some way for a site administrator to log in and make changes. So let's tackle that now.

If you are making a site for the general public, you may want to have things like encrypted password storage (so that even someone with access to the database can't decode them), email-based confirmation and password reset features, automatic logout after some period of inactivity, and so on. Or maybe you want to have people login to your site using their Facebook, Twitter, or Paypal identity. You can get all of these features and more with a plugin called Devise (<https://github.com/plataformatec/devise>). But if you're making a small-scale site to be used with students, it's more likely that you'll want to have full access to the user passwords, and to be able to create logins for users without email accounts. Plus, all the other features would just get in the way. So we're going to make our own user system. It will also help you make sense of a plugin like Devise if and when you use it in the future.

1. The User model

We start by making a User model, with a username, password, real name, and user type (e.g., admin or regular member). We'll also add a timestamp field so we can tell when the user has logged in last. We'll use the scaffold generator so we can get the standard controller actions and views, though we'll be adding a couple more later.

```
rails generate scaffold User username:string password:string  
realname:string usertype:string login_stamp:timestamp
```

(The text above should be all one line.)

Then we rake, to create a "users" table in the database.

```
rake db:migrate
```

Now, if we start up our app and go to the server domain and port plus "/users" in a browser, we can add, view, edit, and delete users just like any other data object. That's a bit too insecure even for us, but we'll take care of that later. For the moment, go ahead and make a user for yourself. Make usertype "admin".

2. Login, logout

The next step is to make login and logout methods. Open the users_controller, and add the following code, just above the very last "end".

```

def login
  # render views/users/login.html.erb
end

def do_login
  if request.post? # Only do if this action has been called with a POST
    request from a form
    user = User.find_by_username_and_password(params[:username],
      params[:password])
    if user
      cookies.permanent.signed[:user_id] = user.id # Set a cookie
        containing the user ID
      user.login_stamp = Time.now
      user.save
      redirect_to(items_url, :notice => 'Login successful')
    else
      redirect_to(:action=>"login", :notice => 'Invalid login name and/or
        password')
    end
  end
end

def logout
  if cookies.signed[:user_id]
    cookies.signed[:user_id] = nil
  end
  redirect_to(items_url, :notice => 'Logout successful')
end

```

The actual logging in is done in the "do_login" action; "login" just calls up a view with a form. Let's make that view now. In the app/views/users directory, make a new file called "login.html.erb", and include the following code:

```

<%= form_tag ([:controller=>"users", :action=>"do_login"]) do %>
  Username:<%= text_field_tag 'username' %>
  Password:<%= password_field_tag 'password' %>
  <%= submit_tag "Log in" %>
<% end %>

```

We need to do one more thing before we can try this out. So far we've only used standard actions (index, show, new, edit, create, update, destroy), and rails knows, for example, if the URL "/users/1" comes in with a GET request, it should invoke the "show" action and pass the parameter "id" with a value of "1". Now that we've added additional actions, though, we need to tell Rails when to invoke them. We do this in **config/routes.rb**.

You'll see a bunch of sample routes, along with

```

resources :users
resources :comments
resources :items

```

These three lines tell Rails to use the default routes with users, comments, and items.

Now we're going to add three lines underneath those (actually, you can put them anywhere before the final "end").

```
    match 'login' => 'users#login'  
    match 'logout' => 'users#logout'  
    match 'users/:action' => 'users'
```

These tell Rails to match a URL with a particular action. The first one says, "if you get a url of '/login', invoke the users controller, login action. The second one is the same but it says to invoke the logout action.

The third line says, "If you get a URL of /users/[action], where [action] can be anything, invoke the users controller and call whatever action is named." This route is a bit of overkill -- after all, we only have one more action ("do_login"), but it's good to see how this works. (After you have saved this, try naming an action that doesn't exist, and see what happens.)

Now try them out. Later we can put in links for logging in and out, but for now call them directly from your browser's address bar. (E.g., "http://rorty.umflint.edu:4001/login")

3. @current_user

So, we've made possible for users to login and log out, but we haven't done anything with that fact. What we need is some way for our app to be able to tell if a user is logged in, and if so, what the user's name and user type is, all at any time.

In order to do that, we will set a global variable called @current_user that holds a user's information when he or she is logged in. Because we want it to be accessible from any part of the site, we'll put the code in the "application controller."

app/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base  
  protect_from_forgery  
  
  before_filter :get_current_user  
  
  def get_current_user  
    if cookies.signed[:user_id]  
      @current_user = User.find(cookies.signed[:user_id])  
    end  
  end  
end
```

The action "get_current_user" checks to see if there is a "user_id" cookie, and if there is, it finds the user with that ID and puts the user object into @current_user.

The line above is a "before_filter," which means that every time an action is called, before doing that action, Rails will do the action "get_current_user." Since it is in the application controller, it will do it regardless of what controller the other action may be in.

So now, in any view or layout, we can add code like this:

```
<% if @current_user %>
  <p>Hi, <%= @current_user.realname %>!</p>
<% end %>
```

(We need the if statement because if the user is not signed in, @current_user will be nil, and Rails will choke when it tries to display one of its attributes.)

To make a login/logout link (displaying the relevant link depending on whether you are logged in), we can do this:

```
<% if @current_user %>
  <%= link_to 'Log out', logout_path %>
<% else %>
  <%= link_to 'Log in', login_path %>
<% end %>
```

4. Current user and comments

Now instead of having an "author" field in the comments form, we can connect each comment to the person who posted it (assuming that we only allow comments to be posted when one is logged in).

First we need to change the users table a bit, adding a "user_id" column and removing the "author" column. At the command line:

```
rails generate migration AddUserIdToComments user_id:integer
rails generate migration RemoveAuthorFromComments author:string
rake db:migrate
```

If the migration name is of the form "AddXXXXToYYYY" or "RemoveXXXXFromYYYY" (YYYY being the name of the table -- in other words, the plural of the name of the model) and is followed by a list of column names and types, Rails will generate migrations with all the necessary information. Sweet!

Challenge!

Add the appropriate "belongs_to" and "has_many" statements in the user and comment models. Then finish changing the comments form and display so that:

- (a) It only shows up if you are logged in (and it shows alternate explanatory text if you are not)
- (b) The current user's id is automatically saved in the "user_id" attribute of the comment.
- (c) The user's real name shows up along with the comments that are displayed.