1. Getting started

Bring up a command line and run:

```
% rails -d sqlite3 squawkr
      create
      create  app/controllers
      create  app/helpers
      create  app/models
      create  app/views/layouts
      create  config/environments
…
```

If you're on a PC, leave off the "-d sqlite3" bit. The instant rails package only comes with MySQL (which happens to be the Rails default as well).

Next, try running the built-in web server.

```
% script/server
=> Booting WEBrick…
=> Rails application started on http://0.0.0.0:3000
=> Ctrl-C to shutdown server; call with --help for options
[2007-03-28 22:39:12] INFO  WEBrick 1.3.1
[2007-03-28 22:39:12] INFO  ruby 1.8.6 (2007-03-13) […]
[2007-03-28 22:39:12] INFO  WEBrick::HTTPServer#start: …
```

You may see alternative messages about something called "mongrel." That's OK. Mongrel is a newer alternative to webrick. Now try pointing your browser to http://localhost:3000/. You should see the default rails screen.

2. The Model

Ask rails to generate some stubs for the model.

```
% script/generate model message
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
      create  app/models/message.rb
      create  test/unit/message_test.rb
      create  test/fixtures/messages.yml
      create  db/migrate
      create  db/migrate/001_create_messages.rb
```

A number of files have been generated. Of particular importance are `app/models/message.rb` (the model itself) and `db/migrate/001_create_messages.rb` (the migration, used to create the table in the database).

Open up the `db/migrate/001_create_messages.rb` file in an editor and change the self.up method to look like this.

```
  def self.up
    create_table :messages do |t|
      t.column :message, :string
      t.column :whom, :string
      t.column :created_on, :datetime
    end
  end
```

This specifies the database in a portable fashion. It's really Ruby being used as a mini-language.

Next, ask rails to create the database.

```
% rake db:migrate
(in /Users/dom/Documents/squawkr)
== CreateMessages: migrating ==
-- create_table(:messages)
   -> 0.0839s
== CreateMessages: migrated (0.0879s) ==
```

You can use the Rails console to play with the model now. It's really useful for experimenting with your objects (and Ruby in general).

```
% script/console
Loading development environment.
>> Message.find(:all)
=> []
>> msg = Message.create(:message => "hello")
=> #<Message:0x30f6ad8 …>
>> msg.message
=> "hello"
>> msg.created_on
=> Fri Mar 30 07:05:10 +0100 2007
```

There are methods on that object for each field in the table. There's also an `id` method. You can change a value by assigning it a value, e.g. `msg.whom = “fred”`. If you change anything, don't forget to call `msg.save` to send your changes back to the database.

Tip: Rails comes with another command, `gem_server`, which serves up documentation through a browser. Try running it, and look at the active_record documentation. Within that, choose the class ActiveRecord::Base to get a list of methods you can call on the object in the console above. Alternatively, visit http://api.rubyonrails.org/ and look at the same class.

Tip: Try running `rake test:unit`. You can see that the generator has also created a minimal test suite for you. Look at `test/unit/message_test.rb` for more.

3. Scaffold

Rails provides a “quick and dirty” CRUD interface. To use it, do:

```
% script/generate scaffold Message
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/messages
      exists  app/views/layouts/
      exists  test/functional/
…
```

The visit http://localhost:3000/messages. You should have an interface to list, add, edit and delete messages. It’s very basic, but is a good start for building on. The main files of interest are `app/controllers/messages_controller.rb` and `app/views/*.rhtml`.

4. Another controller

The scaffold is great, but it's probably overkill for a simple interface like this. Let's start from a fresh perspective.

```
% script/generate controller Squawk
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/squawk
      exists  test/functional/
      create  app/controllers/squawk_controller.rb
      create  test/functional/squawk_controller_test.rb
      create  app/helpers/squawk_helper.rb
```

This gives you a completely blank controller to work in. Let's add an index action (an action is any publically visible method in a controller). The default is "index," just like Apache. Edit `app/controllers/squawk_controller.rb` to look like this:

```
class SquawkController < ApplicationController
  def index
    @messages = Message.find(
      :all,
      :order => 'created_on DESC',
      :limit => 5
    )
  end
end
```

That fetches the five most recent messages and stores them in a field. This is a little bit verbose, and might be better done as a method on the Message class.

Next, we need to edit the view to display these messages. Create a new file, `app/views/squawkr/index.rhtml`, and add this to it.

```
<h1>Recent Messages</h1>
<ul>
    <% for msg in @messages %>
    <li>
        <em><%=h msg.whom %></em>
        said
        <q><%=h msg.message %></q>
    </li>
    <% end %>
</ul>
```

Now you should be able to view this by visiting http://localhost:3000/squawk. There should be a message in there already.

RHTML uses a similar syntax to ASP, JSP and others. <% %> runs code without inserting it into the page. <%= %> runs code and puts it into the page. The "h" in front requests HTML escaping.

Tip: There are no html or body tags here. If you want to add those, create a file `app/layouts/application.rhtml` with them. In the middle where you want the page to appear, say `<% yield %>`.

5. Adding messages

So far so good, but we need a way to add new messages in. Create the file `app/views/squawk/_form.rhtml` and add this to it.

```
<% form_for :message,
            :url => { :action => 'add' } do |f| %>
  <%= f.text_field :whom, :size => 10 %>
  says
  “<%= f.text_field :message %>”
<% end %>
```

This is a “partial” in rails terminology. The name of a partial always starts with an underscore. They can be included from other views, or sometimes used directly (often the case with Ajax).

To use this partial, add the line `<%= render :partial => 'form' %>` to the bottom of `app/views/layout/index.rhtml`. You should see a form appear at the bottom of the page if you refresh the browser window.

In order to make the form work, we need to add another method to `app/controllers/squawk_controller.rb`. Change the file to add in an “add” method.

```
class SquawkController < ApplicationController
  def index
    …
  end
  def add
    @message = Message.new(params[:message])
    if @message.save
      redirect_to :action => "index"
    else
      index
      render :action => "index"
    end
  end
end
```

There’s quite a bit going on here.

- Make a new Message object from the form parameters.
- Try to save that object in the database.
- If it saved correctly, redirect back to the index page.
- If it failed to save, attempt to display the same content as the index page.

That last point is somewhat unusual. The reason is that you need access to the object in order to display validation errors. But to do that, we need to add some validation.

Open up the model, `app/models/message.rb`. Add an extra line so that it looks like this.

```
class Message < ActiveRecord::Base
  validates_presence_of :message, :whom
end
```

That will make rails automatically check that these fields are present and non empty before saving to the database.

Rails can offer some assistance when there are errors. By using `text_field` and other form helpers, each field will automatically get wrapped in `<div class=”fieldWithErrors”></div>`, which you can use CSS to style.

You can also pull out the error messages. To try this, open app/views/squawk/_form.rhtml and add the error_messages_for line.

```
<% form_for :message,
            :url => { :action => 'add' } do |f| %>
  <%= error_messages_for :message %>
  <%= f.text_field :whom, :size => 10 %>
  says
  “<%= f.text_field :message %>”
<% end %>
```

This generates something useful, but ugly. CSS can help as always. But finer grained approaches are also possible.

6. Ajax

The magic buzzword! There are some useful helpers in Rails for performing Ajax where needed. But first, we need to do a little bit of refactoring.

Let's turn the body of the loop in the main view into a partial. Create the file `app/views/squawk/_message.rhtml` with this contents:

```
<li>
  <em><%=h message.whom || 'Somebody' %></em>
  said
  <q><%=h message.message %></q>
</li>
```

Then change `app/views/squawk/index.rhtml` to look like this.

```
<h1>Recent Messages</h1>
<ul id="messages">
  <%= render :partial => 'message',
             :collection => @messages %>
</ul>
<%= render :partial => 'form' %>
```

render has a nice shortcut for rendering the same partial over a list of things.

On to the Ajax. First of all, you need to include the JavaScript libraries that support it. Add this line to the top of `app/views/squawk/index.rhtml`.

```
<%= javascript_include_tag :defaults %>
```

This pulls in Prototype and Script.aculo.us.

Next, change the form to use the Ajax helpers instead of the regular ones.

```
<% remote_form_for :message,
     :update => 'messages',
     :success => 'new Effect.Highlight("messages")',
     :url => { :action => 'add' } do |f| %>
  <%= error_messages_for :message %>
  <%= f.text_field :whom, :size => 10 %>
  says
  “<%= f.text_field :message %>”
<% end %>
```

The key change here is the call to `remote_form_for`. The `:update` says to change the contents of the element with ID “messages” with what comes back from the server. The :success is a bit of JavaScript to run on success. This example runs a yellow fade on the same messages element that we just updated.

To finish off, we need to update the controller so it returns just the contents of the `<ul>`. Open up `app/controllers/squawk_controller.rb` and change the “add” method so it looks like this.

```
class SquawkController < ApplicationController
  def add
    @message = Message.new(params[:message])
    if @message.save
      if request.xhr?
        index
        render :partial => "message",
               :collection => @messages
      else
        redirect_to :action => "index"
      end
    else
      index
      render :action => "index"
    end
  end
end
```

When the request comes from Ajax, `request.xhr?` returns true. In that case, we return a fragment of HTML, which replaces the original in the page.

This is getting rather too large, and needs refactoring. But we have to stop somewhere…