8

Ruby on Rails

Authors

- **Authors**
  - **João Moura Pires (jmp@fct.unl.pt)**
  - **With contributions of**
    - João Costa Seco (jcs@fct.unl.pt)
    - Fernando Birra (fpb@fct.unl.pt)

- **These slides can be freely used for personal or academic matters without permission from the authors, as long as this author list is included.**

- **The use of these slides for commercial matters is not allowed, unless authorized from the authors.**

Bibliography

- **Some examples are extracted or adapted from**
  - Pragmatic Agile Web Development with Rails (4th Edition) by Sam Ruby, Dave Thomas and David Hanson
  - and the book's site
  - http://pragprog.com/
  - Reference material is many times based on http://rubyonrails.org/
  - http://guides.rubyonrails.org/active_record_validations_callbacks.html

Disclaimer

- **This lecture(s) do not cover the Ruby programming language.**
- **See also Recommended readings at the end.**

Table of content

- The Architecture of Rails Applications
- Sample Application: Depot
- Naming conventions
- Model Validations
- Testing
- Catalog Display: Views
- Cart Creation

The Architecture of Rails Applications

MVC architectural pattern

- In **1979**, **Trygve Reenskaug** came up with a new architecture for developing interactive applications. In his design, applications were broken into three types of components: **models**, **views**, and **controllers**.
- The **MVC** architecture was originally intended for conventional GUI applications, where developers found the separation of concerns led to far less coupling, which in turn made the code easier to write and maintain. Each concept or action was expressed in just one well-known place.
- See on http://en.wikipedia.org/wiki/Model-View-Controller
  - Make the distinction between the MVC architectural pattern and the frameworks that follow that pattern
  - check the list of available **web based** frameworks

Models, Views, and Controllers

- The **model** is responsible for maintaining the state of the application. Sometimes this state is transient, lasting for just a couple of interactions with the user. Sometimes the state is permanent and will be stored outside the application, often in a database.
- A model is more than just data; **it enforces all the business rules that apply to that data**.
  - For example, if a discount shouldn't be applied to orders of less than $20, the model will enforce the constraint.
  - By putting the implementation of these business rules in the model, we make sure that nothing else in the application can make our data invalid. The model acts as both a gatekeeper and a data store.

Models, Views, and Controllers

- The **view** is responsible for generating a user interface, normally based on data in the model.
  - For example, an online store will have a list of products to be displayed on a catalog screen. This list will be accessible via the model, but it will be a view that accesses the list from the model and formats it for the end user.
  - Although the view may present the user with various ways of inputting data, the view itself never handles incoming data.
  - The view's work is done once the data is displayed. There may well be many views that access the same model data, often for different purposes.

Models, Views, and Controllers

- **Controllers** orchestrate the application.
  - Controllers **receive events** from the outside world (normally user input), interact with the model, and display an appropriate view to the user.

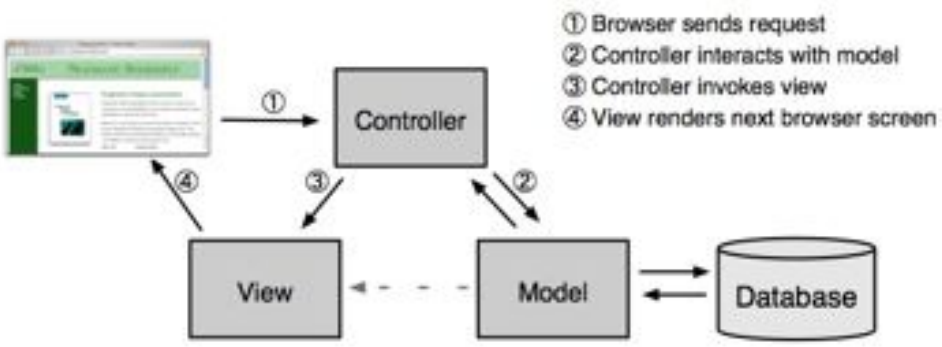


Figure 3.1: The Model-View-Controller architecture

Ruby on Rails is an MVC framework !

- Rails enforces a structure for your application:
  - You develop **models**, **views**, and **controllers** as separate chunks of functionality, and it knits them all together as your program executes.
  - This knitting process is based on the **use of intelligent defaults** so that you typically don't need to write any external configuration metadata to make it all work.

```
rubys> cd work
work> rails new demo
```

```
demo> rails generate controller Say hello goodbye
```

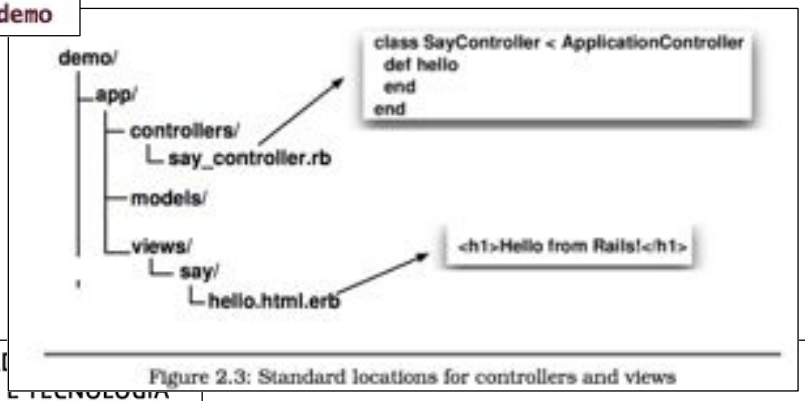


Figure 2.3: Standard locations for controllers and views

Ruby on Rails is an MVC framework !

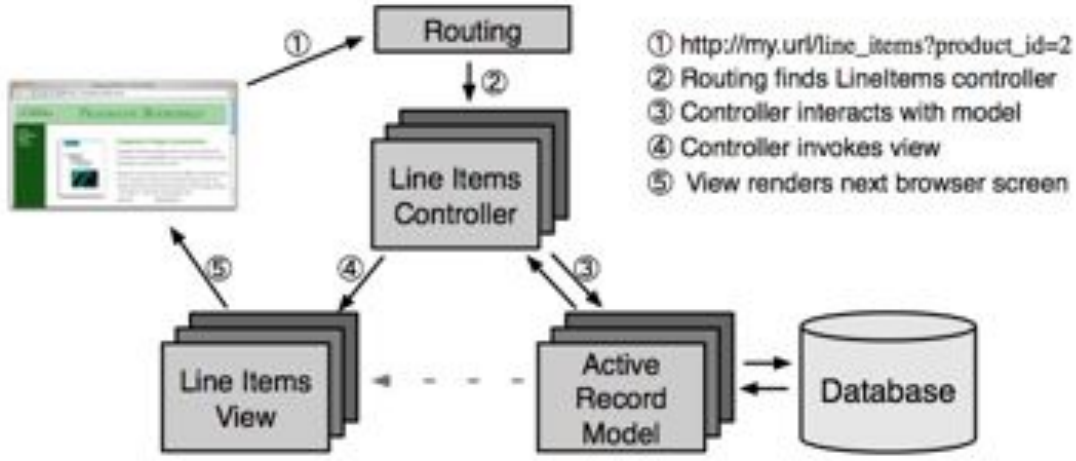


Figure 3.2: Rails and MVC

Ruby on Rails is an MVC framework !

1. An incoming request is first sent to a **router**, which works out **where in the application the request should be sent** and how the request itself should be parsed.

   - Identifies a particular method somewhere in the controller code.

```
Context: the application has previously displayed a product catalog page,
and the user has just clicked the Add to Cart button next to one of the
products.
This button posts to http://localhost:3000/line_items?product_id=2, where
line_items is a resource in our application and 2 is our internal id for
the selected product.
```


Ruby on Rails is an MVC framework !

```
This button posts to http://localhost:3000/line_items?product_id=2, where
line_items is a resource in our application and 2 is our internal id for
the selected product.
```

```
PATH:  line_items?product_id=2
METHOD: POST
=========
CONTROLER: lineItemsController
METHOD: create (since the Method is POST)
ARGUMENT: product_id=2
```

Ruby on Rails is an MVC framework !

2. The create method handles user requests.
3. In this case, it finds the current user's shopping cart (which is an object managed by the model).

   It also asks the model to find the information for product 2.

   It then tells the shopping cart to add that product to itself.
4. Now that the cart includes the new product, we can show it to the user.

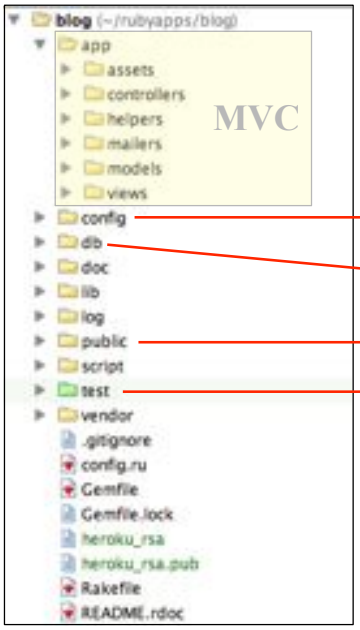
   The controller invokes the view code, but before it does, it arranges things so that the view has access to the cart object from the model.

   In Rails, this invocation is often implicit; again, conventions help link a particular view with a given action.

Sample Application: Blog

Creating a new rails application

```
$ rails new blog
```



Configure your application's runtime rules, routes, database, and more

Contains your current database schema, as well as the database migrations.

The only folder seen to the world as-is. Contains the static files and compiled assets.

Unit tests, fixtures, and other test apparatus.

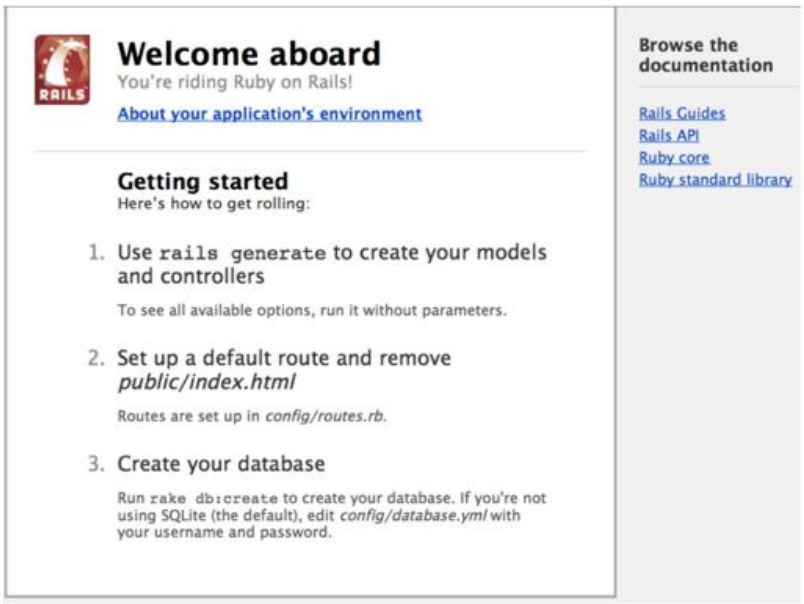
```
$ rake db:create
```

This will create your development and test SQLite3 databases inside the `db/` folder

Running the application

```
$ rails server
```

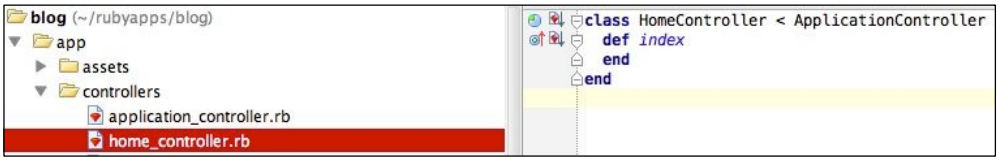
This will fire up an instance of the WEBrick web server by default (Rails can also use several other web servers). To see your application in action, open a browser window and navigate to http://localhost:3000. You should see Rails' default information page:



Hello World!

To get Rails saying "Hello", you need to create at minimum a controller and a view. Fortunately, you can do that in a single command. Enter this command in your terminal:

```
$ rails generate controller home index
```



Hello World!

To get Rails saying "Hello", you need to create at minimum a controller and a view. Fortunately, you can do that in a single command. Enter this command in your terminal:

```
$ rails generate controller home index
```

Hello World!

```
Blog::Application.routes.draw do

  #...
  # You can have the root of your site routed with "root"
  # just remember to delete public/index.html.
  root :to => "home#index"
```

The `root :to => "home#index"` tells Rails to map the root action to the home controller's index action.

Using scafolding for post entity

```
$ rails generate scaffold Post name:string title:string content:text
```

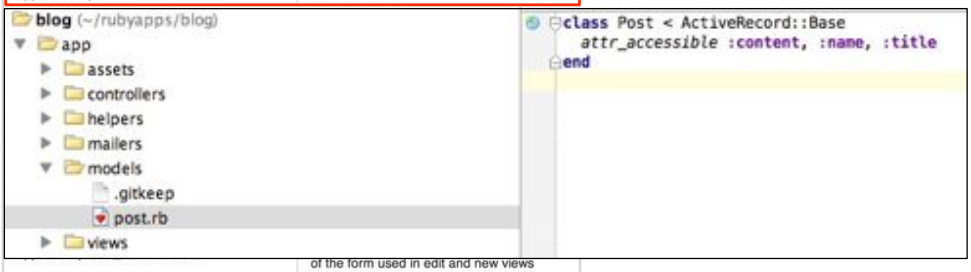
File	Purpose
db/migrate/20100207214725_create_posts.rb	Migration to create the posts table in your database (your name will include a different timestamp)
app/views/posts/_form.html.erb	A partial to control the overall look and feel of the form used in edit and new views
test/functional/posts_controller_test.rb	Functional testing harness for the posts controller
app/helpers/posts_helper.rb	Helper functions to be used from the post views
test/unit/helpers/posts_helper_test.rb	Unit testing harness for the posts helper
app/assets/javascripts/posts.js.coffee	CoffeeScript for the posts controller
app/assets/stylesheets/posts.css.scss	Cascading style sheet for the posts controller
app/assets/stylesheets/scaffolds.css.scss	Cascading style sheet to make the scaffolded views look better


```
class CreatePosts < ActiveRecord::Migration
  def change
    create_table :posts do |t|
      t.string :name
      t.string :title
      t.text :content

      t.timestamps
    end
  end
end
```

Using scafolding for post entity

```
$ rails generate scaffold Post name:string title:string content:text
```

File	Purpose
db/migrate/20100207214725_create_posts.rb	Migration to create the posts table in your database (your name will include a different timestamp)
app/models/post.rb	The Post model
test/functional/posts_controller_test.rb	Functional testing harness for the posts controller
app/helpers/posts_helper.rb	Helper functions to be used from the post views
test/unit/helpers/posts_helper_test.rb	Unit testing harness for the posts helper
app/assets/javascripts/posts.js.coffee	CoffeeScript for the posts controller
app/assets/stylesheets/posts.css.scss	Cascading style sheet for the posts controller
app/assets/stylesheets/scaffolds.css.scss	Cascading style sheet to make the scaffolded views look better



Using scafolding for post entity

```
$ rails generate scaffold Post name:string title:string content:text
```

File	Purpose
db/migrate/20100207214725_create_posts.rb	Migration to create the posts table in your database (your name will include a different timestamp)
app/models/post.rb	The Post model
test/unit/post_test.rb	Unit testing harness for the posts model
app/assets/stylesheets/scaffolds.css.scss	Cascading style sheet to make the scaffolded views look better

FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE NOVA DE LISBOA

8 - DAWeb

Using scafolding for post entity

```
$ rails generate scaffold Post name:string title:string content:text
```

File	Purpose
db/migrate/20100207214725_create_posts.rb	Migration to create the posts table in your database (your name will include a different timestamp)
app/models/post.rb	The Post model
test/unit/post_test.rb	Unit testing harness for the posts model
test/fixtures/posts.yml	Sample posts for use in testing

Fixtures are a way of organizing data that you want to test against; in short, sample data.

Using scafolding for post entity

```
$ rails generate scaffold Post name:string title:string content:text
```

File	Purpose
db/migrate/20100207214725_create_posts.rb	Migration to create the posts table in your database (your name will include a different timestamp)
app/models/post.rb	The Post model
test/unit/post_test.rb	Unit testing harness for the posts model
test/fixtures/posts.yml	Sample posts for use in testing
config/routes.rb	Edited to include routing information for posts


```
Blog::Application.routes.draw do
  resources :posts
```

app/assets/stylesheets/posts.css.scss	Cascading style sheet for the posts controller
app/assets/stylesheets/scaffolds.css.scss	Cascading style sheet to make the scaffolded views look better

Using scafolding for post entity

```
$ rails generate scaffold Post name:string title:string content:text
```

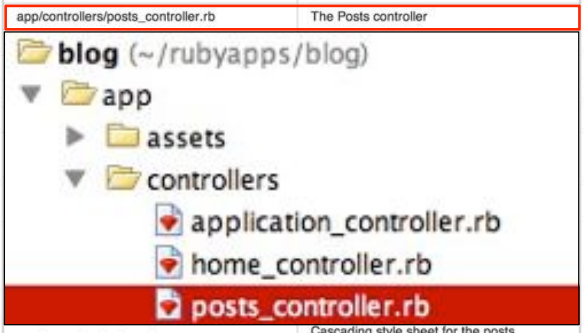
File	Purpose
db/migrate/20100207214725_create_posts.rb	Migration to create the posts table in your database (your name will include a different timestamp)
app/models/post.rb	The Post model
test/unit/post_test.rb	Unit testing harness for the posts model
test/fixtures/posts.yml	Sample posts for use in testing
config/routes.rb	Edited to include routing information for posts
app/controllers/posts_controller.rb	The Posts controller
app/assets/stylesheets/posts.css.scss	Cascading style sheet for the posts controller
app/assets/stylesheets/scaffolds.css.scss	Cascading style sheet to make the scaffolded views look better


```ruby
class PostsController < ApplicationController
  # GET /posts
  # GET /posts.json
  def index ... end

  # GET /posts/1
  # GET /posts/1.json
  def show ... end

  # GET /posts/new
  # GET /posts/new.json
  def new ... end

  # GET /posts/1/edit
  def edit ... end

  # POST /posts
  # POST /posts.json
  def create ... end

  # PUT /posts/1
  # PUT /posts/1.json
  def update ... end

  # DELETE /posts/1
  # DELETE /posts/1.json
  def destroy ... end
end
```


Using scafolding for post entity

$ rails generate scaffold Post name:string title:string content:text

File	Purpose
db/migrate/20100207214725_create_posts.rb	Migration to create the posts table in your database (your name will include a different timestamp)
app/models/post.rb	The Post model
test/unit/post_test.rb	Unit testing harness for the posts model
test/fixtures/posts.yml	Sample posts for use in testing
config/routes.rb	Edited to include routing information for posts
app/controllers/posts_controller.rb	The Posts controller
app/assets/stylesheets/posts.css.scss	Cascading style sheet for the posts controller
app/assets/stylesheets/scaffolds.css.scss	Cascading style sheet to make the scaffolded views look better

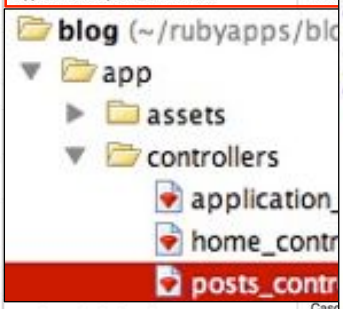



```ruby
class PostsController < ApplicationController
  # GET /posts
  # GET /posts.json
  def index
    @posts = Post.all

    respond_to do |format|
      format.html # index.html.erb
      format.json { render :json => @posts }
    end
  end

  # GET /posts/1
  # GET /posts/1.json
  def show
    @post = Post.find(params[:id])

    respond_to do |format|
      format.html # show.html.erb
      format.json { render :json => @post }
    end
  end
```

Using scafolding for post entity

```
$ rails generate scaffold Post name:string title:string content:text
```

File	Purpose
db/migrate/20100207214725_create_posts.rb	Migration to create the posts table in your data... times
app/models/post.rb	The
test/unit/post_test.rb	Unit
test/fixtures/posts.yml	Sam
config/routes.rb	Edite... posts
app/controllers/posts_controller.rb	The
app/assets/stylesheets/posts.css.scss	Casc... cont...
app/assets/stylesheets/scaffolds.css.scss	Cascading style sheet to make the scaffolded views look better




```
def new
  @post = Post.new

  respond_to do |format|
    format.html # new.html.erb
    format.json { render :json => @post }
  end
end

# GET /posts/1/edit
def edit ... end

# POST /posts
# POST /posts.json
def create
  @post = Post.new(params[:post])

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


Using scafolding for post entity

$ rails generate scaffold Post name:string title:string content:text

File	Purpose
db/migrate/20100207214725_create_posts.rb	Migration to create the posts table in your database (your name will include a different timestamp)
app/models/post.rb	The Post model
test/unit/post_test.rb	Unit testing harness for the posts model
test/fixtures/posts.yml	Sample posts for use in testing
config/routes.rb	Edited to include routing information for posts
app/controllers/posts_controller.rb	The Posts controller
app/views/posts/index.html.erb	A view to display an index of all posts
app/views/posts/edit.html.erb	A view to edit an existing post
app/views/posts/show.html.erb	A view to display a single post
app/views/posts/new.html.erb	A view to create a new post
app/views/posts/_form.html.erb	A partial to control the overall look and feel of the form used in edit and new views
test/functional/posts_controller_test.rb	Functional testing harness for the posts controller
app/helpers/posts_helper.rb	Helper functions to be used from the post views
test/unit/helpers/posts_helper_test.rb	Unit testing harness for the posts helper
app/assets/javascripts/posts.js.coffee	CoffeeScript for the posts controller
app/assets/stylesheets/posts.css.scss	Cascading style sheet for the posts controller
app/assets/stylesheets/scaffolds.css.scss	Cascading style sheet to make the scaffolded views look better

FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE NOVA DE LISBOA

8 - DAWeb

Using scafolding for post entity

```
$ rails generate scaffold Post name:string title:string content:text
```

File	Purpose
db/migrate/20100207214725_create_posts.rb	Migration to create the posts table in your database (your name will include a different timestamp)
app/models/post.rb	Th
test/unit/post_test.rb	Un
test/fixtures/posts.yml	Sa
config/routes.rb	Ed po
app/controllers/posts_controller.rb	Th
app/views/posts/index.html.erb	A v
app/views/posts/edit.html.erb	A v
app/views/posts/show.html.erb	A v
app/views/posts/new.html.erb	A v
app/views/posts/_form.html.erb	A p of
test/functional/posts_controller_test.rb	Fu co
app/helpers/posts_helper.rb	He vie
test/unit/helpers/posts_helper_test.rb	Un
app/assets/javascripts/posts.js.coffee	CoffeeScript for the posts controller
app/assets/stylesheets/posts.css.scss	Cascading style sheet for the posts controller
app/assets/stylesheets/scaffolds.css.scss	Cascading style sheet to make the scaffolded views look better

index.html.erb

```
<h1>Listing posts</h1>

<table>
  <tr>
    <th>Name</th>
    <th>Title</th>
    <th>Content</th>
    <th></th>
    <th></th>
    <th></th>
  </tr>

<% @posts.each do |post| %>
  <tr>
    <td><%= post.name %></td>
    <td><%= post.title %></td>
    <td><%= post.content %></td>
    <td><%= link_to 'Show', post %></td>
    <td><%= link_to 'Edit', edit_post_path(post) %></td>
    <td><%= link_to 'Destroy', post, :method => :delete, :data => { :confirm => 'Are you sure?' } %></td>
  </tr>
<% end %>
</table>

<br />

<%= link_to 'New Post', new_post_path %>
```


Using scafolding for post entity

```
$ rails generate scaffold Post name:string title:string content:text
```

File	Purpose
db/migrate/20100207214725_create_posts.rb	Mi da tim
app/models/post.rb	Th
test/unit/post_test.rb	Un
test/fixtures/posts.yml	Sa
config/routes.rb	Ed po
app/controllers/posts_controller.rb	Th
app/views/posts/index.html.erb	A v
app/views/posts/edit.html.erb	A v
app/views/posts/show.html.erb	A v
app/views/posts/new.html.erb	A v
app/views/posts/_form.html.erb	A p of
test/functional/posts_controller_test.rb	Fu co
app/helpers/posts_helper.rb	He vie
test/unit/helpers/posts_helper_test.rb	Un
app/assets/javascripts/posts.js.coffee	CoffeeScript for the posts controller
app/assets/stylesheets/posts.css.scss	Cascading style sheet for the posts controller
app/assets/stylesheets/scaffolds.css.scss	Cascading style sheet to make the scaffolded views look better

_form.html.erb

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

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

  <div class="field">
    <%= f.label :name %><br />
    <%= f.text_field :name %>
  </div>
  <div class="field">
    <%= f.label :title %><br />
    <%= f.text_field :title %>
  </div>
  <div class="field">
    <%= f.label :content %><br />
    <%= f.text_area :content %>
  </div>
  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>
```

edit.html.erb

```
<h1>Editing post</h1>

<%= render 'form' %>

<%= link_to 'Show', @post %> |
<%= link_to 'Back', posts_path %>
```

FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE NOVA DE LISBOA

8 - DAWeb

Using scafolding for post entity

```
$ rails generate scaffold Post name:string title:string content:text
```

File	Purpose
db/migrate/20100207214725_create_posts.rb	Mi… da… tim…
app/models/post.rb	Th…
test/unit/post_test.rb	Un…
test/fixtures/posts.yml	Sa…
config/routes.rb	Ed… po…
app/controllers/posts_controller.rb	Th…
app/views/posts/index.html.erb	A …
app/views/posts/edit.html.erb	A …
app/views/posts/show.html.erb	A …
app/views/posts/new.html.erb	A …
app/views/posts/_form.html.erb	A … of…
test/functional/posts_controller_test.rb	Fu… co…
app/helpers/posts_helper.rb	He… vie…
test/unit/helpers/posts_helper_test.rb	Un…
app/assets/javascripts/posts.js.coffee	CoffeeScript for the posts controller
app/assets/stylesheets/posts.css.scss	Cascading style sheet for the posts controller
app/assets/stylesheets/scaffolds.css.scss	Cascading style sheet to make the scaffolded views look better

_form.html.erb

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

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

  <div class="field">
    <%= f.label :name %><br />
    <%= f.text_field :name %>
  </div>
  <div class="field">
    <%= f.label :title %><br />
    <%= f.text_field :title %>
  </div>
  <div class="field">
    <%= f.label :content %><br />
    <%= f.text_area :content %>
  </div>
  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>
```

new.html.erb

```
<h1>New post</h1>

<%= render 'form' %>

<%= link_to 'Back', posts_path %>
```


Using scafolding for post entity

```
$ rails generate scaffold Post name:string title:string content:text
```

File	Purpose
db/migrate/20100207214725_create_posts.rb	Migration to create the posts table in your database (your name will include a different timestamp)
app/models/post.rb	The Post model
test/unit/post_test.rb	Unit testing harness for the posts model
test/fixtures/posts.yml	Sample posts for use in testing
config/routes.rb	Edited to include routing information for posts
app/controllers/posts_controller.rb	The Posts controller
app/views/posts/index.html.erb	A view to display an index of all posts
app/views/posts/edit.html.erb	A view to edit an existing post
app/views/posts/show.h…	
app/views/posts/new.h…	
app/views/posts/_form…	
test/functional/posts_c…	
app/helpers/posts_hel…	
test/unit/helpers/posts_…	
app/assets/javascripts/…	
app/assets/stylesheets	
app/assets/stylesheets/scaffolds.css.scss	Cascading style sheet to make the scaffolded views look better

While scaffolding will get you up and running quickly, the code it generates is unlikely to be a perfect fit for your application. You'll most probably want to customize the generated code. Many experienced Rails developers avoid scaffolding entirely, preferring to write all or most of their source code from scratch. Rails, however, makes it really simple to customize templates for generated models, controllers, views and other source files. You'll find more information in the Creating and Customizing Rails Generators & Templates guide.

FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE NOVA DE LISBOA

8 - DAWeb

Using scafolding for post entity

Running a Migration

```
$ rake db:migrate
```

Rails will execute this migration command and tell you it created the Posts table.

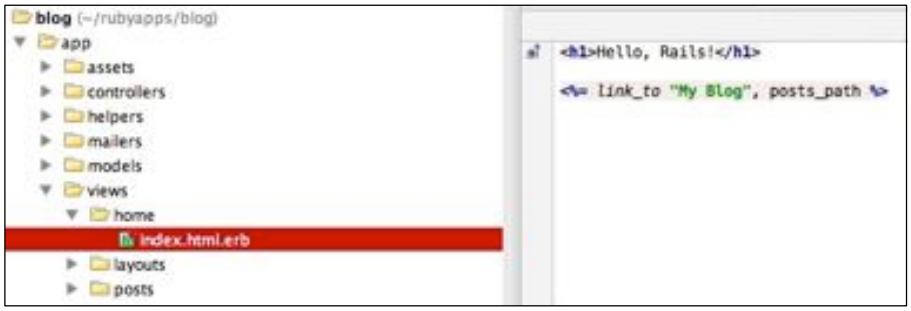
```
Posts: migrating ==============================================
table(:posts)
19s
Posts: migrated (0.0020s) =====================================
```

Because you're working in the development environment by default, this command will apply to the database defined in the development section of your `config/database.yml` file. If you would like to execute migrations in another environment, for instance in production, you must explicitly pass it when invoking the command: `rake db:migrate RAILS_ENV=production`.

Running a Migration

Running a Migration

Linking pages



href="/posts"

The **`link_to`** method is one of Rails' **built-in view helpers**. It creates a hyperlink based on text to display and where to go – in this case, to the path for posts. **posts_path** is a URL helper.

Linking pages

link_to(*args, &block)

Creates a link tag of the given `name` using a URL created by the set of `options`. See the valid options in the documentation for `url_for`. It's also possible to pass a String instead of an options hash, which generates a link tag that uses the value of the String as the href for the link. Using a `:back` Symbol instead of an options hash will generate a link to the referer (a JavaScript back link will be used in place of a referer if none exists). If `nil` is passed as the name the value of the link itself will become the name.

Signatures

```
link_to(body, url, html_options = {})
  # url is a String; you can use URL helpers like
  # posts_path

link_to(body, url_options = {}, html_options = {})
  # url_options, except :confirm or :method,
  # is passed to url_for

link_to(options = {}, html_options = {}) do
  # name
end

link_to(url, html_options = {}) do
  # name
end
```

Options

- `:confirm => 'question?'` - This will allow the unobtrusive JavaScript driver to prompt with the question specified. If the user accepts, the link is processed normally, otherwise no action is taken.
- `:method => symbol of HTTP verb` - This modifier will dynamically create an HTML form and immediately submit the form for processing using the HTTP verb specified. Useful for having links perform a POST operation in dangerous actions like deleting a record (which search bots can follow while spidering your site). Supported verbs are `:post`, `:delete` and `:put`. Note that if the user has JavaScript disabled, the request will fall back to using GET. If `:href => '#'` is used and the user has JavaScript disabled clicking the link will have no effect. If you are relying on the POST behavior, you should check for it in your controller's action by using the request object's methods for `post?`, `delete?` or `put?`.
- `:remote => true` - This will allow the unobtrusive JavaScript driver to make an Ajax request to the URL in question instead of following the link. The drivers each provide mechanisms for listening for the completion of the Ajax request and performing JavaScript operations once they're complete

Linking pages

```
link_to "Profiles", profiles_path
# => <a href="/profiles">Profiles</a>
```

```
link_to "Profile", profile_path(@profile)
# => <a href="/profiles/1">Profile</a>
```

or the even pithier

```
link_to "Profile", @profile
# => <a href="/profiles/1">Profile</a>
```

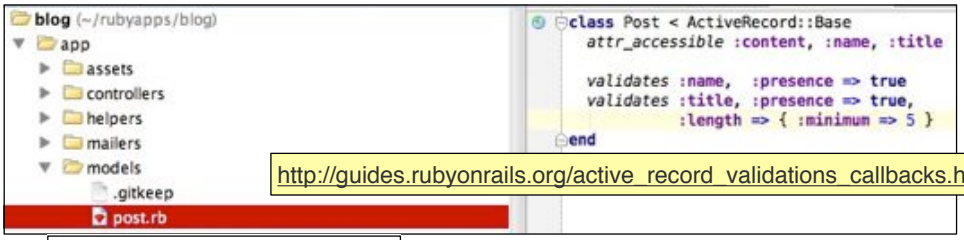
Classes and ids for CSS are easy to produce:

```
link_to "Articles", articles_path, :id => "news", :class => "article"
# => <a href="/articles" class="article" id="news">Articles</a>
```

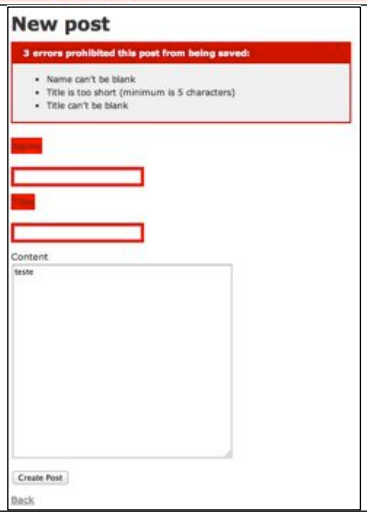
`link_to` can also produce links with anchors or query strings:

```
link_to "Comment wall", profile_path(@profile, :anchor => "wall")
# => <a href="/profiles/i#wall">Comment wall</a>

link_to "Ruby on Rails search", :controller => "searches", :query => "ruby on rails"
# => <a href="/searches?query=ruby+on+rails">Ruby on Rails search</a>

link_to "Nonsense search", searches_path(:foo => "bar", :baz => "quux")
# => <a href="/searches?foo=bar&amp;baz=quux">Nonsense search</a>
```


Adding Some Model Validation



http://guides.rubyonrails.org/active_record_validations_callbacks.html#validations-overview



Understanding how it works: Listing All Posts

routes.rb

```
Blog::Application.routes.draw do
  resources :posts
```

index.html.erb

```
<h1>Listing posts</h1>

<table>
  <tr>
    <th>Name</th>
    <th>Title</th>
    <th>Content</th>
    <th></th>
    <th></th>
    <th></th>
  </tr>

<% @posts.each do |post| %>
  <tr>
    <td><%= post.name %></td>
    <td><%= post.title %></td>
    <td><%= post.content %></td>
    <td><%= link_to 'Show', post %></td>
    <td><%= link_to 'Edit', edit_post_path(post) %></td>
    <td><%= link_to 'Destroy', post, :method => :delete, :data => { :confirm => 'Are you sure?' } %></td>
  </tr>
<% end %>
</table>

<br />

<%= link_to 'New Post', new_post_path %>
```

```
class PostsController < ApplicationController
  # GET /posts
  # GET /posts.json
  def index
    @posts = Post.all

    respond_to do |format|
      format.html # index.html.erb
      format.json { render :json => @posts }
    end
  end
```

`Post.all` returns all of the posts currently in the database as an **array of `Post` records** that we store in an instance variable called `@posts`.

Understanding how it works: Listing All Posts

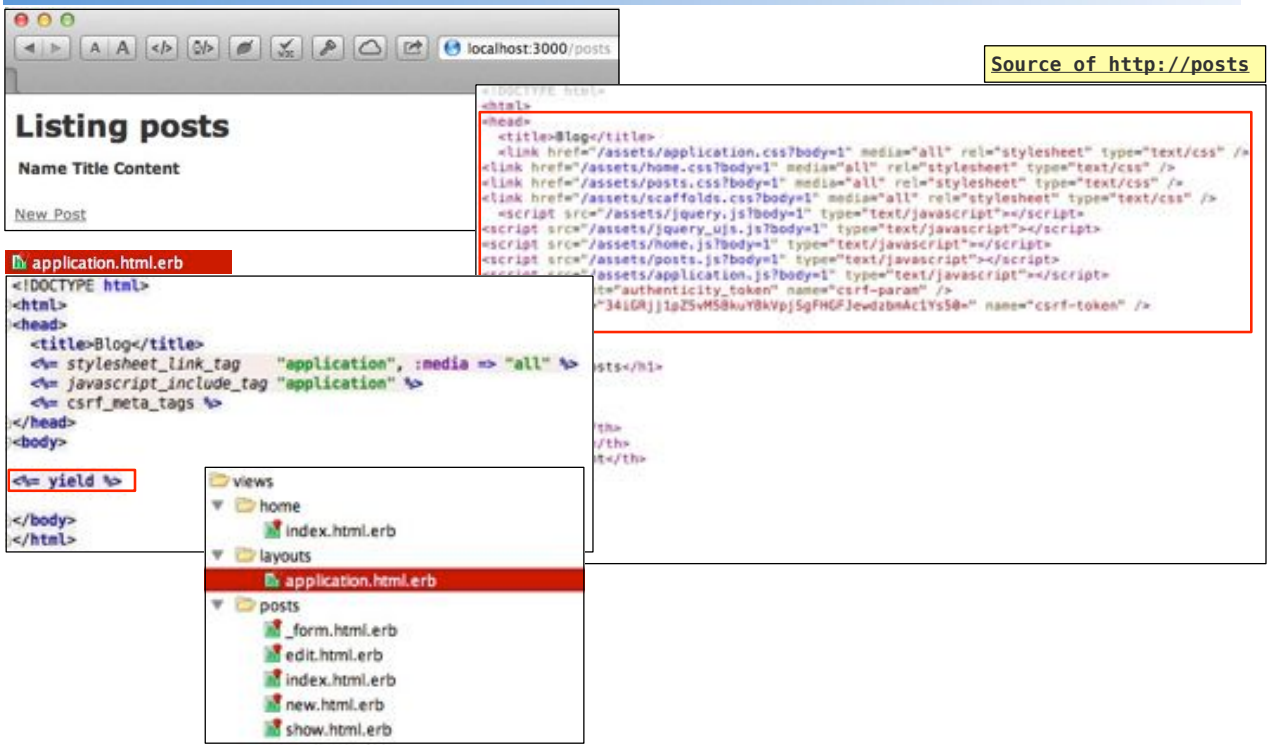
Source of http://posts

```
<!DOCTYPE html>
<html>
<head>
  <title>Blog</title>
  <link href="/assets/application.css?body=1" media="all" rel="stylesheet" type="text/css" />
<link href="/assets/home.css?body=1" media="all" rel="stylesheet" type="text/css" />
<link href="/assets/posts.css?body=1" media="all" rel="stylesheet" type="text/css" />
<link href="/assets/scaffolds.css?body=1" media="all" rel="stylesheet" type="text/css" />
  <script src="/assets/jquery.js?body=1" type="text/javascript"></script>
<script src="/assets/jquery_ujs.js?body=1" type="text/javascript"></script>
<script src="/assets/home.js?body=1" type="text/javascript"></script>
<script src="/assets/posts.js?body=1" type="text/javascript"></script>
<script src="/assets/application.js?body=1" type="text/javascript"></script>
  <meta content="authenticity_token" name="csrf-param" />
<meta content="34iGRjj1pZ5vM58kuY0kVpj5gFHGFlewdzbmAc1Ys50=" name="csrf-token" />
</head>
<body>

<h1>Listing posts</h1>

<table>
  <tr>
    <th>Name</th>
    <th>Title</th>
    <th>Content</th>
    <th></th>
    <th></th>
    <th></th>
  </tr>

</table>
```

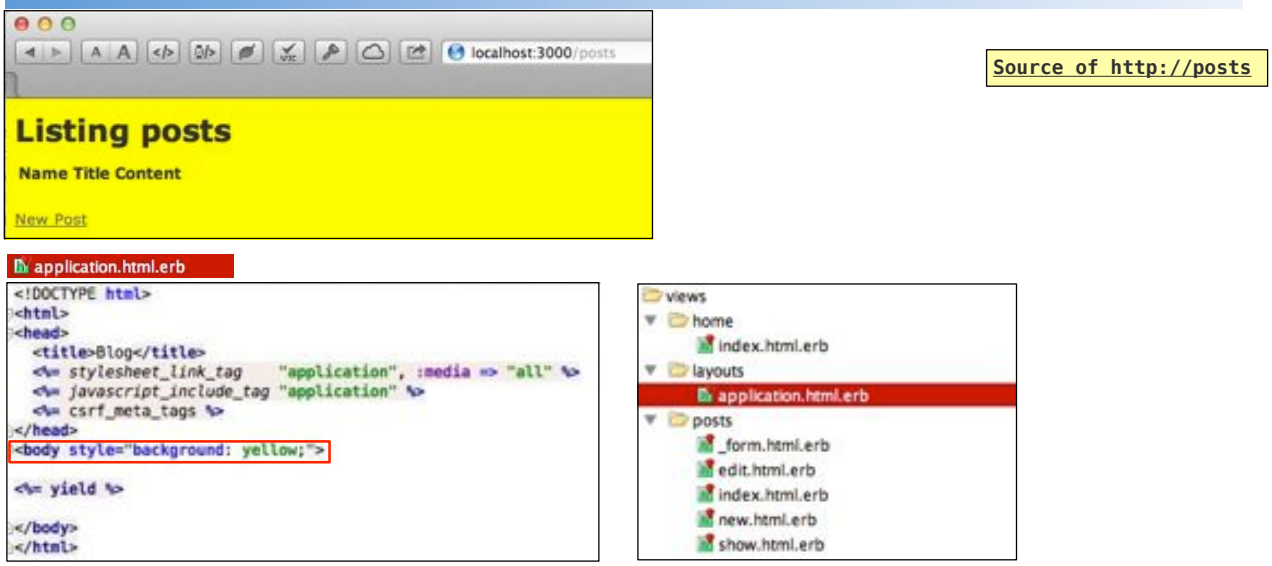
index.html.erb

```
<h1>Listing posts</h1>

<table>
  <tr>
    <th>Name</th>
    <th>Title</th>
    <th>Content</th>
    <th></th>
    <th></th>
    <th></th>
  </tr>

<% @posts.each do |post| %>
  <tr>
    <td><%= post.name %></td>
    <td><%= post.title %></td>
    <td><%= post.content %></td>
    <td><%= link_to 'Show', post %></td>
    <td><%= link_to 'Edit', edit_post_path(post) %></td>
    <td><%= link_to 'Destroy', post, :method => :delete, :data => { :confirm => 'Are you sure?' } %></td>
  </tr>
<% end %>
</table>

<br />

<%= link_to 'New Post', new_post_path %>
```

Understanding how it works: Listing All Posts



Understanding how it works: Listing All Posts



Understanding how it works: Creating New Posts

1 – Presenting a form to enter the data

http://posts/new

posts_controller.rb

```ruby
# GET /posts/new
# GET /posts/new.json
def new
  @post = Post.new

  respond_to do |format|
    format.html # new.html.erb
    format.json { render :json => @post }
  end
end
```

new.html.erb

```
<h1>New post</h1>

<%= render 'form' %>

<%= link_to 'Back', posts_path %>
```

_form.html.erb

```
<%= form_for(@post) do |f| %>
  <% if @post.errors.any? %>
    <div id="error_explanation"...>
  <% end %>

  <div class="field">
    <%= f.label :name %><br />
    <%= f.text_field :name %>
  </div>
  <div class="field">
    <%= f.label :title %><br />
    <%= f.text_field :title %>
  </div>
  <div class="field">
    <%= f.label :content %><br />
    <%= f.text_area :content %>
  </div>
  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>
```

action="/posts" ... method="post">

2 – Create the post with the user supplied data

posts_controller.rb

```ruby
# POST /posts
# POST /posts.json
def create
  @post = Post.new(params[:post])

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


Understanding how it works: Creating New Posts

1 – Presenting a form to enter the data

2 – Create the post with the user supplied data

posts_controller.rb

```ruby
# POST /posts
# POST /posts.json
def create
  @post = Post.new(params[:post])

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


show.html.erb

```
<p id="notice"><%= notice %></p>

<p>
  <b>Name:</b>
  <%= @post.name %>
</p>

<p>
  <b>Title:</b>
  <%= @post.title %>
</p>

<p>
  <b>Content:</b>
  <%= @post.content %>
</p>

<%= link_to 'Edit', edit_post_path(@post) %> |
<%= link_to 'Back', posts_path %>
```

http://posts/1

Post was successfully created.

Name: Name

Title: Title of the Post

Content: Content of the post

Edit | Back

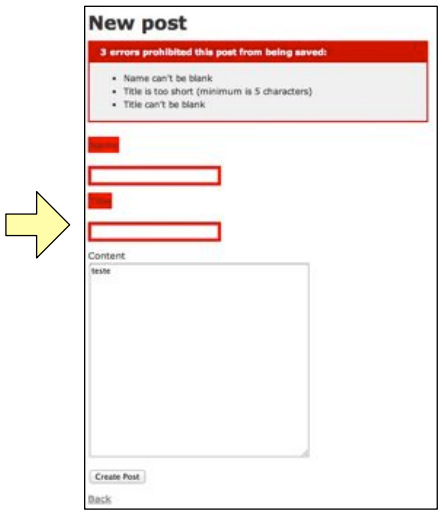
Understanding how it works: Creating New Posts

1 – Presenting a form to enter the data

2 – Create the post with the user supplied data

posts_controller.rb

```ruby
# POST /posts
# POST /posts.json
def create
  @post = Post.new(params[:post])

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



Rails Model Support: "Active Record"

- Map database **tables** to **classes**. If a database has a table called orders, our program will have a class named Order.
- **Rows** correspond to **objects** of the class. A particular order is represented as an object of class Order.
- Within that object, **attributes** are used to get and set the individual **columns**.
- A set of **class-level methods** that perform **table-level operations**. Examples: **find**, **where**, **new**, etc..
- **Instance methods** that perform **operations on the individual rows**. Example: **save.**

Rails Model Support: “Active Record”

- Active Record is the ORM from Rails, which includes:
  - By relying on **convention** and starting with sensible defaults, Active Record **minimizes the amount of configuration** that developers perform. Table and class naming rules, PK and FK attributes, etc.
  - Active Record supports sophisticated validation of model data, and if the form data fails validations, the Rails views can extract and format errors.

View and Controller: Action Pack

- In Rails, the **view** is responsible for creating either all or part of a response to be displayed in a browser, processed by an application or sent as an email.
- In Rails, dynamic content is generated by templates, which come in three flavors. The most common templating scheme, called Embedded Ruby (ERb), embeds snippets of Ruby code within a view document,
- XML Builder can also be used to construct XML documents using Ruby code, the structure of the generated XML will automatically follow the structure of the code.
- Rails also provides RJS views. These allow you to create JavaScript fragments on the server that are then executed on the browser. This is great for creating dynamic Ajax interfaces.

View and Controller: Action Pack

- The Rails controller is the logical center of your application. It coordinates the interaction between the user, the views, and the model. The controller is also home to a number of important ancillary services:
  - It is responsible for **routing external requests to internal actions**. It handles people-friendly URLs extremely well.
  - It manages **caching**, which can give applications orders-of-magnitude performance boosts.
  - It manages **helper modules**, which extend the capabilities of the view templates without bulking up their code.
  - It **manages sessions**, giving users the impression of ongoing interaction with our applications.

Naming Conventions

Naming Conventions

- Ruby conventions
  - **Variable names** where the letters are all **lowercase** and words are separated by **underscores** (ex: `order_status`).
  - **Classes** and **modules** are named differently: there are **no underscores**, and each word in the phrase (including the first) **is capitalized** (ex: `LineItem`).
- Rails conventions
  - **Table names** are like variable names. Rails also assumes that **table names are always plural** (ex: `orders` and `third_parties`).
  - **Files** are named in **lowercase** with **underscores**.

Naming Conventions

- Ex: **class** whose name is **`LineItem`** (Ruby convention). **Rails** would automatically deduce the following:
- That the corresponding database **table** will be called **`line_items`**. That's the class name, converted to lowercase, with underscores between the words and pluralized.
- Rails would also know to look for the **class definition in a file** called **`line_item.rb`** (in the `app/models` directory).

Naming Conventions

- Rails controllers have additional naming conventions. If our application has a `Store` controller, then the following happens:
  - Rails assumes the **class** is called `StoreController` and that it's in a file named `store_controller.rb` in the `app/controllers` directory.
  - It also assumes there's a **helper** module named `StoreHelper` in the file `store_helper.rb` located in the `app/helpers` directory.
  - It will look for **view templates** for this controller in the `app/views/store` directory.
  - It will by default take the output of these views and wrap them in the **layout template** contained in the file `store.html.erb` or `store.xml.erb` in the directory `app/views/layouts`.

Naming Conventions

Model Naming	
Table	line_items
File	app/models/line_item.rb
Class	LineItem

Controller Naming	
URL	http://.../store/list
File	app/controllers/store_controller.rb
Class	StoreController
Method	list
Layout	app/views/layouts/store.html.erb

View Naming	
URL	http://.../store/list
File	app/views/store/list.html.erb (or .builder or .rjs)
Helper	module StoreHelper
File	app/helpers/store_helper.rb

Figure 18.3: How naming conventions work across a Rails application

Naming Conventions

- In normal Ruby code you have to use the `require` keyword to include Ruby source files before you reference the classes and modules in those files.
- Because **Rails knows the relationship between filenames and class names**, require is normally not necessary in a Rails application. Instead, **the first time you reference a class or module that isn't known, Rails uses the naming conventions to convert the class name to a filename** and tries to load that file behind the scenes.

Grouping Controllers into Modules

- Rails does this using a simple naming convention.
  - If an incoming request has a controller named `admin/book`, Rails will look for the controller called `book_controller.rb` in the directory `app/controllers/admin`.
  - Imagine that our program has two such groups of controllers (say, admin/xxx and content/xxx) and that both groups define a book controller. There'd be a file called `book_controller.rb` in **both the admin and content subdirectories** of `app/controllers`. **If Rails took no further steps, these two classes would clash.**

the book controller in the admin subdirectory would be declared like this:

```
class Admin::BookController < ActionController::Base
  # ...
end
```

The book controller in the content subdirectory would be in the Content module:

```
class Content::BookController < ActionController::Base
  # ...
end
```

Grouping Controllers into Modules

- Imagine that our program has two such groups of controllers (say, admin/xxx and content/xxx) and that both groups define a book controller. There'd be a file called `book_controller.rb` in **both the admin and content subdirectories** of `app/controllers`. **If Rails took no further steps, these two classes would clash.**
- The templates for these controllers appear in subdirectories of `app/views`. Thus, the view template corresponding to this request:
  - http://my.app/admin/book/edit/1234
- will be in this file:
  - `app/views/admin/book/edit.html.erb`

```
myapp> rails generate controller Admin::Book action1 action2 ...
```


Rails Model Support: "Active Record"

Rails Model Support: “Active Record”

- **Active Record** is the **object-relational mapping** (ORM) layer supplied with Rails. It is the part of Rails that implements your **application’s model**.
  - Map database **tables** to **classes**;
  - **Rows** correspond to **objects** of the class;
  - Within that object, **attributes** are used to get and set the individual **columns**.
  - A set of **class-level methods** that perform **table-level operations**. Examples: find, where, new, etc..
  - **Instance methods** that perform **operations on the individual rows**. Example: save.
- By relying on **convention** and starting with sensible defaults, Active Record minimizes the amount of configuration that developers perform. Table and class naming rules, PK and FK attributes, etc.

Naming Conventions

- Ruby conventions
  - **Variable names** where the letters are all **lowercase** and words are separated by **underscores** (ex: `order_status`).
  - **Classes** and **modules** are named differently: there are **no underscores**, and each word in the phrase (including the first) **is capitalized** (ex: `LineItem`).
- Rails conventions
  - **Table names** are like variable names. Rails also assumes that **table names are always plural** (ex: `orders` and `third_parties`).
  - **Files** are named in **lowercase** with **underscores**.

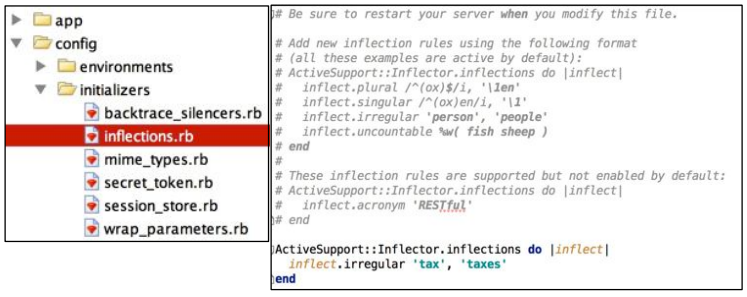
Naming Conventions

- Ex: **class** whose name is `LineItem` (Ruby convention). **Rails** would automatically deduce the following:
  - That the corresponding database **table** will be called `line_items`. That's the class name, converted to lowercase, with underscores between the words and pluralized.
  - Rails would also know to look for the **class definition in a file** called `line_item.rb` (in the `app/models` directory).

Class Name	Table Name	Class Name	Table Name
Order	orders	LineItem	line_items
TaxAgency	tax_agencies	Person	people
Batch	batches	Datum	data
Diagnosis	diagnoses	Quantity	quantities

Naming Conventions: Special cases

- You can add to Rails' understanding of the idiosyncrasies and inconsistencies of the English language by modifying the inflection file provided:



- If you have legacy tables you have to deal with, you can control the table name associated with a given model by setting the **table_name** for a given class:

```
class Sheep < ActiveRecord::Base
  self.table_name = "sheep"
end
```

Rails Model Support: generating a model

- **`rails new app`**
  - creates all the necessary folders and file to start your application
- **`rake db:create`**
  - creates your development and test SQLite3 databases inside the db/ folder
- **`rails generate model Student`**

  `name:string student_number:integer status:string foto_url:string`
  - creates:
    - a migration: `db/migrate/20121117210433_create_students.rb` — Migration
    - a model: `app/models/student.rb` — Model
    - a test unit folder with:
      - `test/unit/student_test.rb` — Unit Tests
      - `test/fixtures/students.yml` — Fixtures

Rails Model Support: generating a model

- **`rails generate scaffold Student`**

  `name:string student_number:integer status:string foto_url:string`
  - creates all the previous stuff and controller and views to respond to the CRUD operations

Migration

```
class CreateStudents < ActiveRecord::Migration
  def change
    create_table :students do |t|
      t.string :name
      t.integer :student_number
      t.string :status
      t.string :foto_url

      t.timestamps
    end
  end
end
```

Model

```
class Student < ActiveRecord::Base
  attr_accessible :foto_url, :name, :status, :student_number
end
```

Fixtures

```
one:
  name: MyString
  student_number: 1
  status: MyString
  foto_url: MyString

two:
  name: MyString
  student_number: 1
  status: MyString
  foto_url: MyString
```

Unit Tests

```
require 'test_helper'

class StudentTest < ActiveSupport::TestCase
  # test "the truth" do
  #   assert true
  # end
end
```

Migrations

- **Migrations are a convenient way to alter the database in a structured and organised manner, without editing SQL.**
- **Active Record tracks which migrations have already been run so all you have to do is update your source and run rake db:migrate.**
- **It will also update your db/schema.rb file to match the structure of your database.**
- **Migrations also allow you to describe these transformations using Ruby.**
- **The great thing about this is that (like most of Active Record's functionality) it is database independent.**
- **For example you could use SQLite3 in development, but MySQL in production.**

Migrations are ruby classes

```
class CreateProducts < ActiveRecord::Migration
  def up
    create_table :products do |t|
      t.string :name
      t.text :description

      t.timestamps
    end
  end

  def down
    drop_table :products
  end
end
```

- Adds a table called products with a string column called name and a text column called description;
- A primary key column called id will also be added, however since this is the default we do not need to ask for this;
- The timestamp columns created_at and updated_at which Active Record populates automatically will also be added;
- Reversing this migration is as simple as dropping the table.

Migrations

Rails 3.1 makes migrations smarter by providing a new change method. This method is preferred for writing constructive migrations (adding columns or tables). The migration knows how to migrate your database and reverse it when the migration is rolled back without the need to write a separate down method.

```
class CreateProducts < ActiveRecord::Migration
  def change
    create_table :products do |t|
      t.string :name
      t.text :description

      t.timestamps
    end
  end
end
```


Migrations

Active Record provides methods that perform common data definition tasks in a database independent way (you'll read about them in detail later):

- add_column
- add_index
- change_column
- change_table
- create_table
- drop_table
- remove_column
- remove_index
- rename_column

Migrations: supported data types

Active Record supports the following database column types:

- `:binary`
- `:boolean`
- `:date`
- `:datetime`
- `:decimal`
- `:float`
- `:integer`
- `:primary_key`
- `:string`
- `:text`
- `:time`
- `:timestamp`

Migrations: supported data types

	db2	mysql	openbase	oracle
:binary	blob(32768)	blob	object	blob
:boolean	decimal(1)	tinyint(1)	boolean	number(1)
:date	date	date	date	date
:datetime	timestamp	datetime	datetime	date
:decimal	decimal	decimal	decimal	decimal
:float	float	float	float	number
:integer	int	int(11)	integer	number(38)
:string	varchar(255)	varchar(255)	char(4096)	varchar2(255)
:text	clob(32768)	text	text	clob
:time	time	time	time	date
:timestamp	timestamp	datetime	timestamp	date

Migrations: supported data types

	postgresql	sqlite	sqlserver	sybase
:binary	bytea	blob	image	image
:boolean	boolean	boolean	bit	bit
:date	date	date	date	datetime
:datetime	timestamp	datetime	datetime	datetime
:decimal	decimal	decimal	decimal	decimal
:float	float	float	float(8)	float(8)
:integer	integer	integer	int	int
:string	(note 1)	varchar(255)	varchar(255)	varchar(255)
:text	text	text	text	text
:time	time	datetime	time	time
:timestamp	timestamp	datetime	datetime	timestamp

Migrations: creating a migration

- **The model and scaffold generators will create migrations appropriate for adding a new model.**

```
rails generate model NAME [field[:type][:index] field[:type][:index]] [options]
```

  - By default, the generated migration will include t.timestamps (which creates the `updated_at` and `created_at` columns that are automatically populated by Active Record)

- **Creating a Standalone Migration.**

```
$ rails generate migration AddPartNumberToProducts
```

This will create an empty but appropriately named migration:

```
class AddPartNumberToProducts < ActiveRecord::Migration
  def change
  end
end
```

Migrations: creating a migration

- **Creating a Standalone Migration**
  - If the migration name is of the form “AddXXXToYYY” or “RemoveXXXFromYYY” and is followed by a list of column names and types then a migration containing the appropriate add_column and remove_column statements will be created.

```
$ rails generate migration AddPartNumberToProducts part_number:string
```

will generate

```
class AddPartNumberToProducts < ActiveRecord::Migration
  def change
    add_column :products, :part_number, :string
  end
end
```


Migrations: creating a migration

- **Creating a Standalone Migration**
  - If the migration name is of the form “AddXXXToYYY” or “RemoveXXXFromYYY” and is followed by a list of column names and types then a migration containing the appropriate add_column and remove_column statements will be created.

```
$ rails generate migration RemovePartNumberFromProducts part_number:string
```

generates

```
class RemovePartNumberFromProducts < ActiveRecord::Migration
  def up
    remove_column :products, :part_number
  end

  def down
    add_column :products, :part_number, :string
  end
end
```

Migrations: running migrations

- **`rake db:migrate`.**
  - **In its most basic form it just runs the up or change method for all the migrations that have not yet been run. If there are no such migrations, it exits. It will run these migrations in order based on the date of the migration.**
  - **Note that running the db:migrate also invokes the db:schema:dump task, which will update your db/schema.rb file to match the structure of your database.**
- **`rake db:rollback`**
  - **This will run the down method from the latest migration. If you need to undo several migrations you can provide a STEP parameter:**

```
$ rake db:rollback STEP=3
```


Migrations and the Schema.db

- **`rails generate model Student`**

  `name:string student_number:integer status:string foto_url:string`

Migration

```
class CreateStudents < ActiveRecord::Migration
  def change
    create_table :students do |t|
      t.string :name
      t.integer :student_number
      t.string :status
      t.string :foto_url

      t.timestamps
    end
  end
end
```

Model

```
class Student < ActiveRecord::Base
  attr_accessible :foto_url, :name, :status, :student_number
end
```

- `rake db:migrate`

```
# This file is auto-generated from the current state of the database. Instead
# of editing this file, please use the migrations feature of Active Record to
# incrementally modify your database, and then regenerate this schema definition.
#
# Note that this schema.rb definition is the authoritative source for your
# database schema. If you need to create the application database on another
# system, you should be using db:schema:load, not running all the migrations
# from scratch. The latter is a flawed and unsustainable approach (the more migrations
# you'll amass, the slower it'll run and the greater likelihood for issues).
#
# It's strongly recommended to check this file into your version control system.

ActiveRecord::Schema.define(:version => 20121117210433) do

  create_table "students", :force => true do |t|
    t.string   "name"
    t.integer  "student_number"
    t.string   "status"
    t.string   "foto_url"
    t.datetime "created_at",     :null => false
    t.datetime "updated_at",     :null => false
  end

end
```

Migrations and the Schema.db

- `rake db:migrate`

```
# This file is auto-generated from the current state of the database. Instead
# of editing this file, please use the migrations feature of Active Record to
# incrementally modify your database, and then regenerate this schema definition.
#
# Note that this schema.rb definition is the authoritative source for your
# database schema. If you need to create the application database on another
# system, you should be using db:schema:load, not running all the migrations
# from scratch. The latter is a flawed and unsustainable approach (the more migrations
# you'll amass, the slower it'll run and the greater likelihood for issues).
#
# It's strongly recommended to check this file into your version control system.

ActiveRecord::Schema.define(:version => 20121117210433) do

  create_table "students", :force => true do |t|
    t.string   "name"
    t.integer  "student_number"
    t.string   "status"
    t.string   "foto_url"
    t.datetime "created_at",     :null => false
    t.datetime "updated_at",     :null => false
  end

end
```

- `rails console`

```
Loading development environment (Rails 3.2.9.rc2)

>> Student.column_names

=> ["id", "name", "student_number", "status", "foto_url", "created_at", "updated_at"]
```


Migrations and the Schema.db

- `rails console`

```
Loading development environment (Rails 3.2.9.rc2)

>> Student.column_names

=> ["id", "name", "student_number", "status", "foto_url", "created_at", "updated_at"]

>> Student.columns_hash["status"]

=> #<ActiveRecord::ConnectionAdapters::SQLiteColumn:0x10fa807e0 @primary=false,

@scale=nil, @default=nil, @sql_type="varchar(255)", @coder=nil, @name="status",

@limit=255, @type=:string, @precision=nil, @null=true>
```

Further reading about migrations on

http://guides.rubyonrails.org/migrations.html

Chapter 23 of Agile Web Development with Rails (4th Edition)

Active Record Associations

The purpose of Active Record Associations

- Expressiveness
- They make common operations simpler and easier in your code. Consider a simple Rails application that includes a model for customers and a model for orders. Each customer can have many orders.
  - Without associations:
    - Model

```
class Customer < ActiveRecord::Base
end

class Order < ActiveRecord::Base
end
```

    - Add a new order for an existing customer

```
@order = Order.create(:order_date => Time.now,
  :customer_id => @customer.id)
```

    - Deleting a customer, and ensuring that all of its orders get deleted as well

```
@orders = Order.where(:customer_id => @customer.id)
@orders.each do |order|
  order.destroy
end
@customer.destroy
```

The purpose of Active Record Associations

- With Active Record Associations:
  - **Model**

```ruby
class Customer < ActiveRecord::Base
  has_many :orders, :dependent => :destroy
end

class Order < ActiveRecord::Base
  belongs_to :customer
end
```

  - **Add a new order for an existing customer**

```ruby
@order = @customer.orders.create(:order_date => Time.now)
```

  - **Deleting a customer, and ensuring that all of its orders get deleted as well**

```ruby
@customer.destroy
```

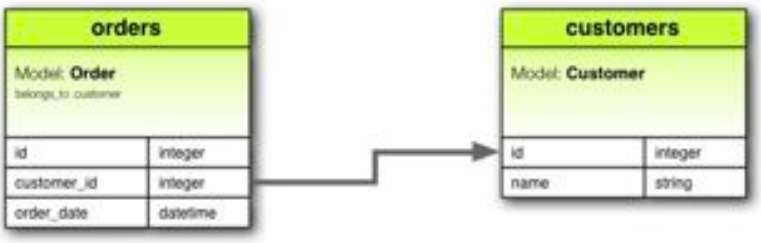

The Types of Associations

- An association is a connection between two Active Record models.
- Rails supports six types of associations
  - **belongs_to**
  - **has_one**
  - **has_many**
  - **has_many :through**
  - **has_one :through**
  - **has_and_belongs_to_many**

Rails types of associations: belongs_to

- Each instance of the declaring model “belongs to” one instance of the other model.

```
class Order < ActiveRecord::Base
  belongs_to :customer
end
```



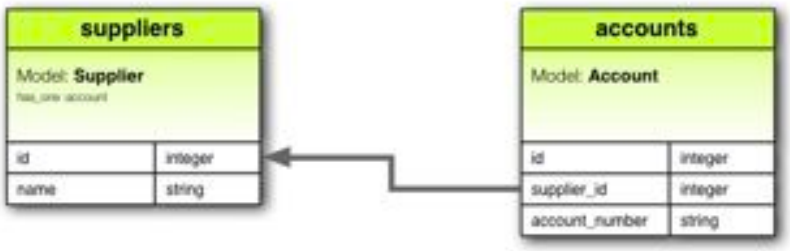

```
class Order < ActiveRecord::Base
  belongs_to :customer
end
```

- A foreign key is placed on the origin model. The name of the foreign key is the name of the destination model followed by _id

Rails types of associations: has_one

- Each instance of a model contains or possesses one instance of another model.

```
class Supplier < ActiveRecord::Base
  has_one :account
end
```



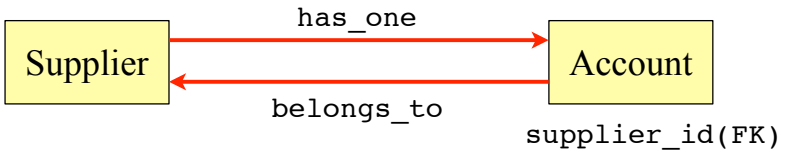

```
class Supplier < ActiveRecord::Base
  has_one :account
end
```

- A foreign key is placed on the destination model.

Choosing Between belongs_to and has_one

- `one-to-one` relationship between two models.
- The `has_one` relationship says that one of something is yours – that is, that something points back to you. For example, it makes more sense to say that a supplier owns an account than that an account owns a supplier.

```
class Supplier < ActiveRecord::Base
  has_one :account
end

class Account < ActiveRecord::Base
  belongs_to :supplier
end
```



Rails types of associations: has_many

- This association indicates that each instance of the model has **zero or more instances** of another model. You'll often find this association on the "other side" of a belongs_to association

```
class Customer < ActiveRecord::Base
  has_many :orders
end
```

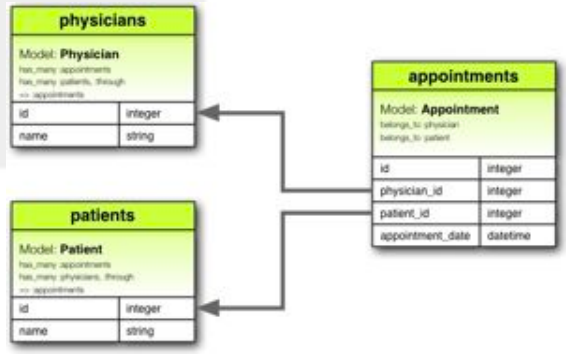
The name of the other model is pluralized when declaring a has_many association.


```
class Customer < ActiveRecord::Base
  has_many :orders
end
```

Rails types of associations: has_many :through

- A `has_many :through` association is often used to set up a many-to-many connection with another model. This association indicates that the declaring model can be matched with zero or more instances of another model by proceeding through a third model.

```
class Physician < ActiveRecord::Base
  has_many :appointments
  has_many :patients, :through => :appointments
end

class Appointment < ActiveRecord::Base
  belongs_to :physician
  belongs_to :patient
end

class Patient < ActiveRecord::Base
  has_many :appointments
  has_many :physicians, :through => :appointments
end
```



Rails types of associations: has_many :through

- The `has_many :through` association is also useful for setting up “shortcuts” through nested has_many associations. For example, if a document has many sections, and a section has many paragraphs, you may sometimes want to get a simple collection of all paragraphs in the document.

```
class Document < ActiveRecord::Base
  has_many :sections
  has_many :paragraphs, :through => :sections
end

class Section < ActiveRecord::Base
  belongs_to :document
  has_many :paragraphs
end

class Paragraph < ActiveRecord::Base
  belongs_to :section
end
```

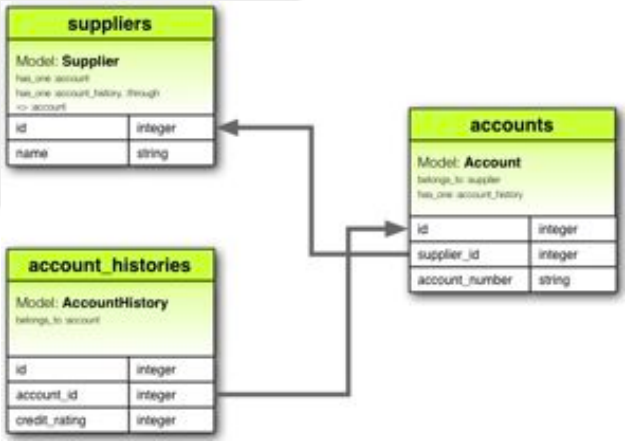
With `:through => :sections` specified, Rails will now understand:

```
@document.paragraphs
```

Rails types of associations: `has_one :through`

- A `has_one :through` association sets up a one-to-one connection with another model. This association indicates that the declaring model can be matched with one instance of another model by proceeding through a third model.

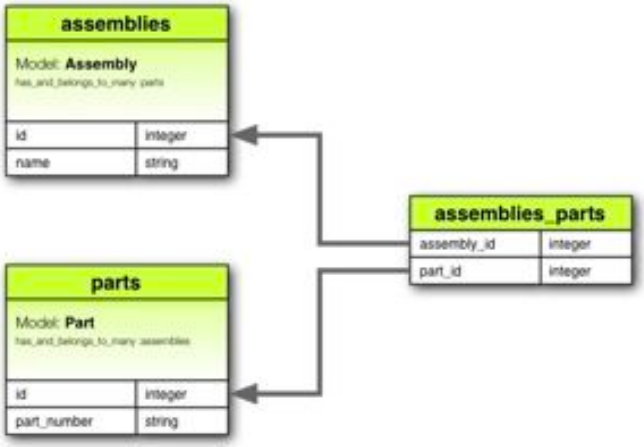
```
class Supplier < ActiveRecord::Base
  has_one :account
  has_one :account_history, :through => :account
end

class Account < ActiveRecord::Base
  belongs_to :supplier
  has_one :account_history
end

class AccountHistory < ActiveRecord::Base
  belongs_to :account
end
```



Types of associations: `has_and_belongs_to_many`

- A `has_and_belongs_to_many` association creates a direct many-to-many connection with another model, with no intervening model.

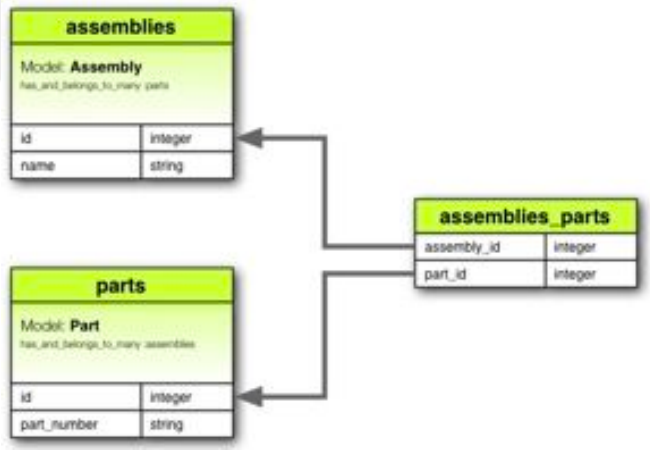
```
class Assembly < ActiveRecord::Base
  has_and_belongs_to_many :parts
end

class Part < ActiveRecord::Base
  has_and_belongs_to_many :assemblies
end
```



Between has_many :through and has_and_belongs_to_many

- Rails offers **two different ways** to declare a **many-to-many relationship** between models. The **simpler** way is to use `has_and_belongs_to_many`, which allows you to make the association directly

```
class Assembly < ActiveRecord::Base
  has_and_belongs_to_many :parts
end

class Part < ActiveRecord::Base
  has_and_belongs_to_many :assemblies
end
```



Between has_many :through and has_and_belongs_to_many

- The second way to declare a many-to-many relationship is to use has_many :through. This makes the association indirectly, through a join model:

```
class Assembly < ActiveRecord::Base
  has_many :manifests
  has_many :parts, :through => :manifests
end

class Manifest < ActiveRecord::Base
  belongs_to :assembly
  belongs_to :part
end

class Part < ActiveRecord::Base
  has_many :manifests
  has_many :assemblies, :through => :manifests
end
```

- You should set up a `has_many :through` relationship if you need to work with the relationship model as an independent entity. If you don't need to do anything with the relationship model, it may be simpler to set up a `has_and_belongs_to_many` relationship.
- You should use has_many :through if you need validations, callbacks, or extra attributes on the join model.

Polymorphic Associations

- With polymorphic associations, a model can belong to more than one other model, on a single association.
- For example, you might have a **picture model** that belongs to **either** an **employee** model or a **product** model. Here's how this could be declared:

```
class Picture < ActiveRecord::Base
  belongs_to :imageable, :polymorphic => true
end

class Employee < ActiveRecord::Base
  has_many :pictures, :as => :imageable
end

class Product < ActiveRecord::Base
  has_many :pictures, :as => :imageable
end
```

- From an instance of the Employee model, you can retrieve a collection of pictures: `@employee.pictures`.
- Similarly, you can retrieve `@product.pictures`.

Polymorphic Associations

- If you have an instance of the Picture model, you can get to its parent via @picture.imageable. To make this work, you need to declare both a **foreign key column** and a **type column** in the model that declares the polymorphic interface:

```
class CreatePictures < ActiveRecord::Migration
  def change
    create_table :pictures do |t|
      t.string  :name
      t.integer :imageable_id
      t.string  :imageable_type
      t.timestamps
    end
  end
end
```

- This migration can be simplified by using the t.references form:

```
class CreatePictures < ActiveRecord::Migration
  def change
    create_table :pictures do |t|
      t.string :name
      t.references :imageable, :polymorphic => true
      t.timestamps
    end
  end
end
```

Self Joins

- A model that should have a relation to itself.
- For example, you may want to store all employees in a single database model, but be able to trace relationships such as between manager and subordinates

```
class Employee < ActiveRecord::Base
  has_many :subordinates, :class_name => "Employee",
    :foreign_key => "manager_id"
  belongs_to :manager, :class_name => "Employee"
end
```

- You can retrieve `@employee.subordinates` and `@employee.manager`.

Readings on Active Record Associations

- http://guides.rubyonrails.org/association_basics.html
-

Active Record Query Interface

Retrieving Objects from the Database

- Active Record provides several finder methods. Each finder method allows you to pass arguments:
  - `where`
  - `select`
  - `group`
  - `order`
  - `reorder`
  - `reverse_order`
  - `limit`
  - `offset`
  - `joins`
  - `includes`
  - `lock`
  - `readonly`
  - `from`
  - `having`
- All of the above methods return an instance of ActiveRecord::Relation.
- The primary operation of `Model.find(options)` can be summarized as:
  - Convert the supplied options to an equivalent SQL query.
  - Fire the SQL query and retrieve the corresponding results from the database.
  - Instantiate the equivalent Ruby object of the appropriate model for every resulting row.
  - Run `after_find` callbacks, if any.

Retrieving a Single Object

- Using a Primary Key

```ruby
# Find the client with primary key (id) 10.
client = Client.find(10)
# => #<Client id: 10, first_name: "Ryan">
```

- First

```ruby
client = Client.first
# => #<Client id: 1, first_name: "Lifo">
```

- Last

```ruby
client = Client.last
# => #<Client id: 221, first_name: "Russel">
```

- First!

  `Model.first!` raises `RecordNotFound` if no matching record is found.

- Last!

  `Model.last!` raises `RecordNotFound` if no matching record is found.

Retrieving Multiple Objects

- Using Multiple Primary Keys
  - `Model.find(array_of_primary_key)` accepts an array of primary keys, returning an array containing all of the matching records for the supplied primary keys.

```ruby
# Find the clients with primary keys 1 and 10.
client = Client.find([1, 10]) # Or even Client.find(1, 10)
# => [#<Client id: 1, first_name: "Lifo">, #<Client id: 10, first_name: "Ryan">]
```

  - `Model.find(array_of_primary_key)` will raise an `ActiveRecord::RecordNotFound` exception unless a matching record is found for all of the supplied primary keys.

Retrieving Multiple Objects

- Retrieving Multiple Objects in Batches
  - Motivation

```ruby
# This is very inefficient when the users table has thousands of rows.
User.all.each do |user|
  NewsLetter.weekly_deliver(user)
end
```

  - find_each

```ruby
User.find_each do |user|
  NewsLetter.weekly_deliver(user)
end
```

retrieves a batch of records and then yields ***each*** record to the block individually as a model

  - find_in_batches

```ruby
# Give add_invoices an array of 1000 invoices at a time
Invoice.find_in_batches(:include => :invoice_lines) do |invoices|
  export.add_invoices(invoices)
end
```

retrieves a batch of records and then yields *the* ***entire batch*** to the block as an array of models.

Retrieving Multiple Objects

- Find_each
  - The find_each method retrieves a batch of records and then yields each record to the block individually as a model.

```ruby
User.find_each do |user|
  NewsLetter.weekly_deliver(user)
end
```

  - will retrieve 1000 records (the current default for both find_each and find_in_batches) and then yield each record individually to the block as a model.
  - This process is repeated until all of the records have been processed
  - Options for find_each
    - :batch_size
    - :start

```ruby
User.find_each(:start => 2000, :batch_size => 5000) do |user|
  NewsLetter.weekly_deliver(user)
end
```

      - By default, records are fetched in ascending order of the primary key

Conditions

- The where method allows you to specify conditions to limit the records returned, representing the WHERE-part of the SQL statement. Conditions can either be specified as a string, array, or hash.
  - Pure String Conditions
    - Building your own conditions as pure strings can leave you vulnerable to SQL injection exploits.

```
Client.where("orders_count = '2'")
```

  - **Array Conditions**
    - A query string with “?” and an array of values to be used in the placeholders “?”
  - **Hash Conditions**
    - With hash conditions, you pass in a hash with keys of the fields you want conditionalised and the values of how you want to conditionalise them

Conditions: array conditions

- Active Record will go through the first element in the conditions value and any additional elements will replace the question marks (?) in the first element.

```
Client.where("orders_count = ?", params[:orders])
```

- If you want to specify multiple conditions:

```
Client.where("orders_count = ? AND locked = ?", params[:orders], false)
```

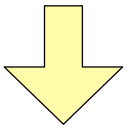
- **Placeholder Conditions**
  - Instead of using ? you can also specify keys/values hash in your array conditions:

```
Client.where("created_at >= :start_date AND created_at <= :end_date",
             {:start_date => params[:start_date], :end_date => params[:end_date]})
```

Conditions: array conditions

- **Range Conditions**
  - You can use the conditions option coupled with the BETWEEN SQL statement

```
Client.where(:created_at => (params[:start_date].to_date)..(params[:end_date].to_date))
```



```
SELECT "clients".* FROM "clients"
WHERE ("clients"."created_at" BETWEEN '2010-09-29' AND '2010-11-30')
```


Conditions: hash conditions

- Active Record also allows you to pass in hash conditions which can increase the readability of your conditions syntax. With hash conditions, you pass in a **hash with keys of the fields you want conditionalised and the values of how you want to conditionalise** them:
  - Equality Conditions

```
Client.where(:locked => true)

Client.where('locked' => true)
```

  - Range Conditions

```
Client.where(:created_at => (Time.now.midnight - 1.day)..Time.now.midnight)
```

  - Subset Conditions

```
Client.where(:orders_count => [1,3,5])
```

Ordering

- To retrieve records from the database in a specific order, you can use the order method.

```
Client.order("created_at DESC")
# OR
Client.order("created_at ASC")
```

Or ordering by multiple fields:

```
Client.order("orders_count ASC, created_at DESC")
```


Selecting Specific Fields

- By default, Model.find selects all the fields from the result set using select *. To select only a subset of fields from the result set, you can specify the subset via the select method. For example, to select only `viewable_by` and `locked` columns:

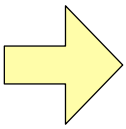
```
Client.select("viewable_by, locked")
```

- Be careful because this also means you're initializing a model object with only the fields that you've selected.
- If the select method is used, all the returning objects will be read only.
- If you would like to only grab a single record per unique value in a certain field, you can use `uniq`:

```
Client.select(:name).uniq
```

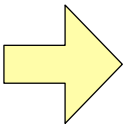
Limit and Offset

```
Client.limit(5)
```



```
SELECT * FROM clients LIMIT 5
```

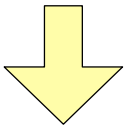
```
Client.limit(5).offset(30)
```



```
SELECT * FROM clients LIMIT 5 OFFSET 30
```


Group

```
Order.select("date(created_at) as ordered_date,
              sum(price) as total_price").group("date(created_at)")
```

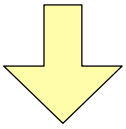


```
SELECT date(created_at) as ordered_date,
       sum(price) as total_price
FROM orders
GROUP BY date(created_at)
```

Joining Tables

- Active Record provides a finder method called joins for specifying JOIN clauses on the resulting SQL. There are multiple ways to use the joins method.
  - Using a String SQL Fragment

```
Client.joins('LEFT OUTER JOIN addresses ON addresses.client_id = clients.id')
```



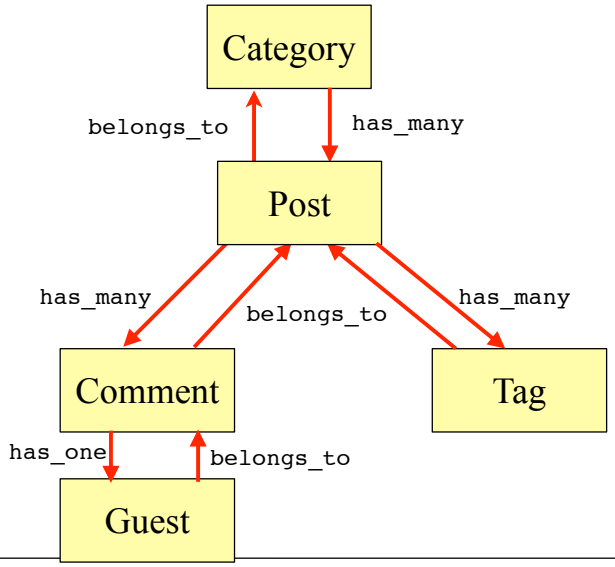
```
SELECT clients.*
FROM clients LEFT OUTER JOIN addresses ON addresses.client_id = clients.id
```

  - Using Array/Hash of Named Associations
    - Active Record lets you use the names of the associations defined on the model as a shortcut for specifying JOIN clause for those associations when using the joins method.

Joining Tables: Using Array/Hash of Named Associations

- Using Array/Hash of Named Associations
  - Active Record lets you use the names of the associations defined on the model as a shortcut for specifying JOIN clause for those associations when using the joins method.

```
class Category < ActiveRecord::Base
  has_many :posts
end

class Post < ActiveRecord::Base
  belongs_to :category
  has_many :comments
  has_many :tags
end

class Comment < ActiveRecord::Base
  belongs_to :post
  has_one :guest
end

class Guest < ActiveRecord::Base
  belongs_to :comment
end

class Tag < ActiveRecord::Base
  belongs_to :post
end
```

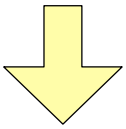


Joining Tables: Using Array/Hash of Named Associations

- **Joining a Single Association**
  - Return a Category object for all categories with posts

```
Category.joins(:posts)
```

```
class Category < ActiveRecord::Base
  has_many :posts
end
```



```
SELECT categories.*
FROM categories
INNER JOIN posts ON posts.category_id = categories.id
```

  - Note that you will see **duplicate** categories if more than one post has the same category.

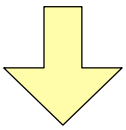
```
Category.joins(:post).select("distinct(categories.id)")
```


Joining Tables: Using Array/Hash of Named Associations

- **Joining Multiple Associations**
  - Return all posts that have a category and at least one comment

```
Post.joins(:category, :comments)
```

```
class Post < ActiveRecord::Base
  belongs_to :category
  has_many :comments
  has_many :tags
end
```



```
SELECT posts.*
FROM posts
INNER JOIN categories ON posts.category_id = categories.id
INNER JOIN comments ON comments.post_id = posts.id
```

  - Note again that posts with multiple comments will show up multiple times.

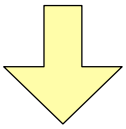
```
Post.joins(:category, :comments).select("distinct(posts.id)")
```

Joining Tables: Using Array/Hash of Named Associations

- **Joining Nested Associations (Single Level)**
  - Return all posts that have a comment made by a guest

```
class Post < ActiveRecord::Base
  belongs_to :category
  has_many :comments
  has_many :tags
end

class Comment < ActiveRecord::Base
  belongs_to :post
  has_one :guest
end
```

```
Post.joins(:comments => :guest)
```

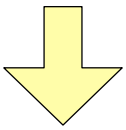


```
SELECT posts.*
FROM posts
INNER JOIN comments ON comments.post_id = posts.id
INNER JOIN guests ON guests.comment_id = comments.id
```


Joining Tables: Using Array/Hash of Named Associations

- **Joining Nested Associations (Multiple Level)**
  - Return all posts that have a comment made by a guest

```
Category.joins(:posts => [{:comments => :guest}, :tags])
```



```
SELECT categories.*
FROM categories
INNER JOIN posts ON posts.category_id = categories.id
INNER JOIN comments ON comments.post_id = posts.id
INNER JOIN guests ON guests.comment_id = comments.id
INNER JOIN tags ON tags.post_id = posts.id
```

Joining Tables: Specifying Conditions on the Joined Tables

- **You can specify conditions on the joined tables using the regular Array and String conditions.**
- **Hash conditions provides a special syntax for specifying conditions for the joined tables**

```
time_range = (Time.now.midnight - 1.day)..Time.now.midnight

Client.joins(:orders).where('orders.created_at' => time_range)
```

```
time_range = (Time.now.midnight - 1.day)..Time.now.midnight

Client.joins(:orders).where(:orders => {:created_at => time_range})
```


Eager Loading Associations

- **Consider the following code, which finds 10 clients and prints their postcodes:**

```
clients = Client.limit(10)

clients.each do |client|
  puts client.address.postcode
end
```

- **The above code executes 1 (to find 10 clients) + 10 (one per each client to load the address) = 11 queries in total.**

```
clients = Client.includes(:address).limit(10)

clients.each do |client|
  puts client.address.postcode
end
```

- **The above code will execute just 2 queries, as opposed to 11 queries in the previous case**

Eager Loading Multiple Associations

- **Active Record lets you eager load any number of associations with a single Model.find call by using an array, hash, or a nested hash of array/hash with the includes method**
  - **Array of Multiple Associations**

```
Post.includes(:category, :comments)
```

This loads all the posts and the associated category and comments for each post.

  - **Nested Associations Hash**

```
Category.includes(:posts => [{:comments => :guest}, :tags]).find(1)
```

This will find the category with id 1 and eager load all of the associated posts, the associated posts' tags and comments, and every comment's guest association

Scopes

- **Scoping allows you to specify commonly-used ARel queries which can be referenced as method calls on the association objects or models;**
- **With these scopes, you can use every method previously covered such as where, joins and includes;**
- **All scope methods will return an ActiveRecord::Relation object which will allow for further methods (such as other scopes) to be called on it.**

```
class Post < ActiveRecord::Base
  scope :published, where(:published => true)
  scope :published_and_commented, published.and(self.arel_table[:comments_count].gt(0))
end
```

- **We can call a scope on either the class or on an association consisting of Post objects**

```
Post.published # => [published posts]
```

```
category = Category.first
category.posts.published # => [published posts belonging to this category]
```

Dynamic Finders

- For every field (also known as an attribute) you define in your table, Active Record provides a finder method. If you have a field called `first_name` on your Client model for example, you get `find_by_first_name` and `find_all_by_first_name` for free from Active Record
- You can specify an exclamation point (!) on the end of the dynamic finders to get them to raise an ActiveRecord::RecordNotFound error if they do not return any records.
- If you have a `locked` field on the Client model, you also get `find_by_locked` and `find_all_by_locked` methods. If you want to find both by name and locked, you can chain these finders together by simply typing “and” between the fields

```
Client.find_by_first_name_and_locked("Ryan", true)
```


Model Validations

The big picture for Validation

- During the normal operation of a Rails application, objects may be created, updated, and destroyed. **Active Record** provides hooks into this object life cycle so that you can control your application and its data.
- Validations allow you to ensure that **only valid data is stored in your database**. **Callbacks** and **observers** allow you to **trigger logic before** or **after** an alteration of an object's state.
- There are several ways to validate data before it is saved into your database, including **native database constraints**, **client-side** validations, **controller-level** validations, and **model-level** validations.

Validation by Database constraints

- Database constraints and/or stored procedures make the validation mechanisms database-dependent and can make testing and maintenance more difficult.
- However, if your database is used by other applications, it may be a good idea to use some constraints at the database level.
- Additionally, database-level validations can safely handle some things (such as uniqueness in heavily-used tables) that can be difficult to implement otherwise.

Validation by Client-side validations

- Client-side validations can be useful, but are generally unreliable if used alone.
  - If they are implemented using JavaScript, they may be bypassed if JavaScript is turned off in the user's browser.
- However, if combined with other techniques, client-side validation can be a convenient way to provide users with immediate feedback as they use your site.

Validation at Controller-level or Model-level

- **Controller-level** validations can be tempting to use, but often become unwieldy and difficult to test and maintain. Whenever possible, it's a **good idea to keep your controllers skinny**, as it will make your application a pleasure to work with in the long run.
- **Model-level validations are the best way to ensure that only valid data is saved into your database**. They are database agnostic, cannot be bypassed by end users, and are **convenient to test and maintain**. Rails makes them easy to use, provides **built-in helpers** for common needs, and **allows you to create your own validation methods** as well.

When Does Validation Happen?

- Two kinds of Active Record objects: those that correspond to a **row inside your database** and **those that do not**. When you create a fresh object, for example using the new method, that object does not belong to the database yet

- Creating and saving a new record will send an SQL INSERT operation to the database. Updating an existing record will send an SQL UPDATE operation instead. **Validations are typically run before these commands are sent to the database**. If any validations fail, the object will be marked as invalid and Active Record will not perform the INSERT or UPDATE operation

When Does Validation Happen?

The following methods trigger validations, and will save the object to the database only if the object is valid:

- `create`
- `create!`
- `save`
- `save!`
- `update`
- `update_attributes`
- `update_attributes!`

The bang versions (e.g. `save!`) raise an exception if the record is invalid. The non-bang versions don't: `save` and `update_attributes` return `false`, `create` and `update` just return the objects.

valid? and invalid?

To verify whether or not an object is valid, Rails uses the `valid?` method. You can also use this method on your own. `valid?` triggers your validations and returns true if no errors were added to the object, and false otherwise.

```
class Person < ActiveRecord::Base
  validates :name, :presence => true
end

Person.create(:name => "John Doe").valid? # => true
Person.create(:name => nil).valid? # => false
```


errors[ ]

To verify whether or not a particular attribute of an object is valid, you can use `errors[:attribute]`. It returns an array of all the errors for `:attribute`. If there are no errors on the specified attribute, an empty array is returned.

This method is only useful *after* validations have been run, because it only inspects the errors collection and does not trigger validations itself. It's different from the `ActiveRecord::Base#invalid?` method explained above because it doesn't verify the validity of the object as a whole. It only checks to see whether there are errors found on an individual attribute of the object.

```
class Person < ActiveRecord::Base
  validates :name, :presence => true
end

>> Person.new.errors[:name].any? # => false
>> Person.create.errors[:name].any? # => true
```

Validation

- No product should be allowed in the database if it has an **empty title** or **description field**, an **invalid URL for the image**, or an **invalid price**.
  - `validates :title, :description, :image_url, :presence => true`

3 errors prohibited this product from being saved:

- Title can't be blank
- Description can't be blank
- Image url can't be blank

Validation

- We'd also like to validate that the price is a valid, positive number.
  - `validates :price, :numericality => {:greater_than_or_equal_to => 0.01}`

Validation

- Each product has a unique title.
  - `validates :title, :uniqueness => true`

- URL entered for the image is valid.
  -
```
validates :image_url, :format => {
  :with    => %r{\.(gif|jpg|png)$}i,
  :message => 'must be a URL for GIF, JPG or PNG image.'
}
```

The image URL looks reasonable.

Model Validation

```
class Product < ActiveRecord::Base
  validates :title, :description, :image_url, :presence => true
  validates :price, :numericality => {:greater_than_or_equal_to => 0.01}
  validates :title, :uniqueness => true
  validates :image_url, :format => {
    :with    => %r{\.(gif|jpg|png)$}i,
    :message => 'must be a URL for GIF, JPG or PNG image.'
  }
end
```

Validation Helpers

- **acceptance**: Validates that a checkbox on the user interface was checked when a form was submitted. This is typically used when the user needs to agree to your application's terms of service, confirm reading some text, or any similar concept. This validation is very specific to web applications and this 'acceptance' does not need to be recorded anywhere in your database (if you don't have a field for it, the helper will just create a virtual attribute).

- **validates_associated**: You should use this helper when your model has associations with other models and they also need to be validated.

```
class Library < ActiveRecord::Base
  has_many :books
  validates_associated :books
end
```


Validation Helpers

- **confirmation**: You should use this helper when you have two text fields that should receive exactly the same content. For example, you may want to confirm an email address or a password. This validation creates a virtual attribute whose name is the name of the field that has to be confirmed with "_confirmation" appended.

```
class Person < ActiveRecord::Base
  validates :email, :confirmation => true
end
```

In your view template you could use something like

```
<%= text_field :person, :email %>
<%= text_field :person, :email_confirmation %>
```

Validation Helpers

- **exclusion**: This helper validates that the attributes' values are not included in a given set. In fact, this set can be any enumerable object.

```
class Account < ActiveRecord::Base
  validates :subdomain, :exclusion => { :in => %w(www us ca jp),
    :message => "Subdomain %{value} is reserved." }
end
```

- **format:** This helper validates the attributes' values by testing whether they match a given regular expression, which is specified using the :with option.

```
class Product < ActiveRecord::Base
  validates :legacy_code, :format => { :with => /\A[a-zA-Z]+\z/,
    :message => "Only letters allowed" }
end
```


Validation Helpers

- **inclusion**: This helper validates that the attributes' values are included in a given set. In fact, this set can be any enumerable object.

```
class Coffee < ActiveRecord::Base
  validates :size, :inclusion => { :in => %w(small medium large),
    :message => "%{value} is not a valid size" }
end
```

- **length:** This helper validates the length of the attributes' values. It provides a variety of options, so you can specify length constraints in different ways.

```
class Person < ActiveRecord::Base
  validates :name, :length => { :minimum => 2 }
  validates :bio, :length => { :maximum => 500 }
  validates :password, :length => { :in => 6..20 }
  validates :registration_number, :length => { :is => 6 }
end
```

Validation Helpers

- **numericality**: This helper validates that your attributes have only numeric values. By default, it will match an optional sign followed by an integral or floating point number. To specify that only integral numbers are allowed set :only_integer to true.

```
class Player < ActiveRecord::Base
  validates :points, :numericality => true
  validates :games_played, :numericality => { :only_integer => true }
end
```

- other contraints:
  - :greater_than, :greater_than_or_equal_to,
  - :equal_to,
  - :less_than, :less_than_or_equal_to,
  - :odd, :even

Validation Helpers

- **presence**: This helper validates that the specified attributes are not empty. It uses the blank? method to check if the value is either nil or a blank string, that is, a string that is either empty or consists of whitespace.

- If you want to be sure that an association is present, you'll need to test whether the foreign key used to map the association is present, and not the associated object itself.

```
class LineItem < ActiveRecord::Base
  belongs_to :order
  validates :order_id, :presence => true
end
```

- Since false.blank? is true, if you want to validate the presence of a boolean field you should use validates :field_name, :inclusion => { :in => [true, false] }.

Validation Helpers

- **uniqueness**: This helper validates that the attribute's value is unique right before the object gets saved. It does not create a uniqueness constraint in the database, so it may happen that two different database connections create two records with the same value for a column that you intend to be unique. To avoid that, you must create a unique index in your database.

There is a `:scope` option that you can use to specify other attributes that are used to limit the uniqueness check:

```
class Holiday < ActiveRecord::Base
  validates :name, :uniqueness => { :scope => :year,
    :message => "should happen once per year" }
end
```

There is also a `:case_sensitive` option that you can use to define whether the uniqueness constraint will be case sensitive or not. This option defaults to true.

```
class Person < ActiveRecord::Base
  validates :name, :uniqueness => { :case_sensitive => false }
end
```


Validation Helpers

- **validates_with**: This helper passes the record to a separate class for validation.

```
class Person < ActiveRecord::Base
  validates_with GoodnessValidator
end

class GoodnessValidator < ActiveModel::Validator
  def validate(record)
    if record.first_name == "Evil"
      record.errors[:base] << "This person is evil"
    end
  end
end
```

- **validates_each**: This helper validates attributes against a block. It doesn't have a predefined validation function. You should create one using a block, and every attribute passed to validates_each will be tested against it.

```
class Person < ActiveRecord::Base
  validates_each :name, :surname do |model, attr, value|
    model.errors.add(attr, 'must start with upper case') if value =~ /\
  end
end
```

Other topics on validation

- **Check on:**
  - **http://guides.rubyonrails.org/active_record_validations_callbacks.html**
- Common Validation Options: `:allow_nil, :allow_blank, :on`
- Conditional Validation
- Performing Custom Validations
- Working with Validation Errors
- Displaying Validation Errors in the View
- **Callbacks**
- **Observers**

Other topics on validation

- **Callbacks**
  - Callbacks are methods that get called at certain moments of an object's life cycle. With callbacks it's possible to write code that will run whenever an Active Record object is created, saved, updated, deleted, validated, or loaded from the database.
- **Observers**
  - Observers are similar to callbacks, but with important differences. Whereas callbacks can pollute a model with code that isn't directly related to its purpose, observers allow you to add the same functionality outside of a model. For example, it could be argued that a User model should not include code to send registration confirmation emails. **Whenever you use callbacks with code that isn't directly related to your model, you may want to consider creating an observer instead**.

Sample Application: Depot

Application overview

- Two different roles or actors: the **buyer** and the **seller**:
- The **buyer** uses Depot to browse the products we have to sell, select some to purchase, and supply the information needed to create an order.
- The **seller** uses Depot to maintain a list of products to sell, to determine the orders that are awaiting shipping, and to mark orders as shipped.

Page Flow: buyer

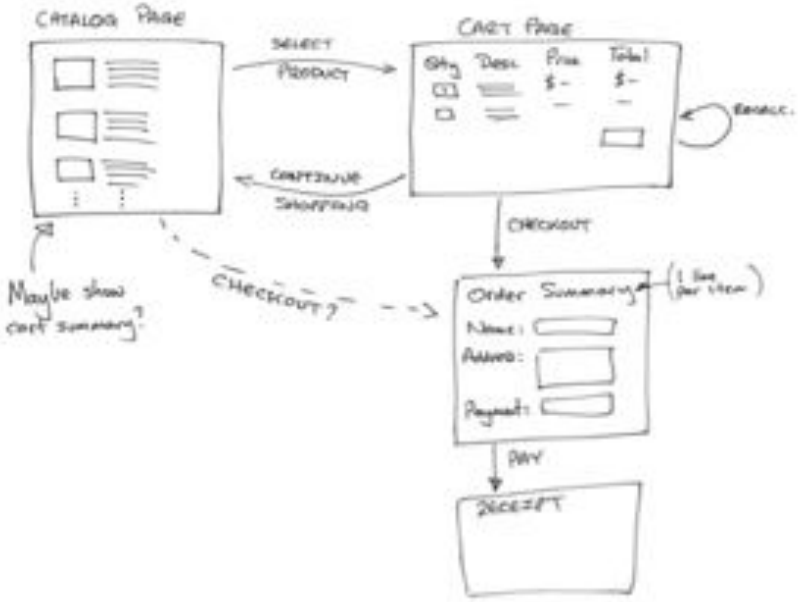


Figure 5.1: Flow of buyer pages

Page Flow: seller

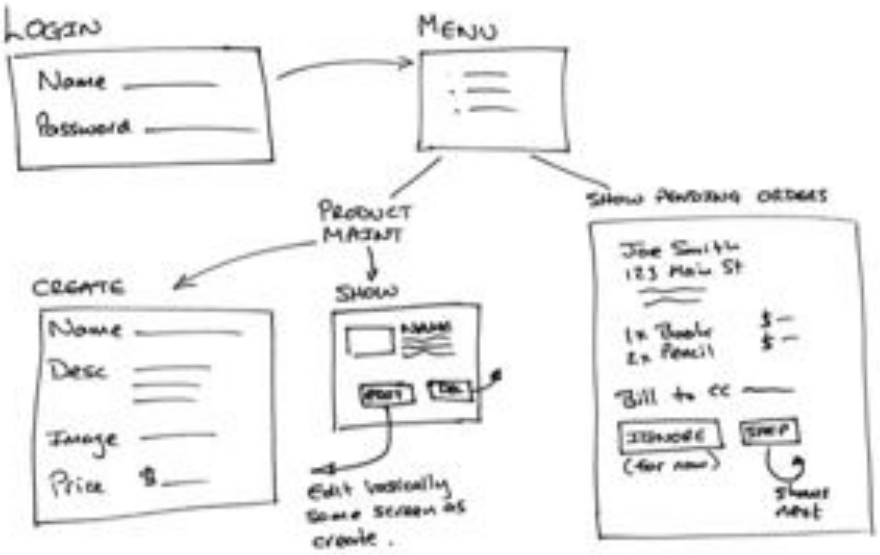


Figure 5.2: Flow of seller pages

Data: Product, order, buyer and seller

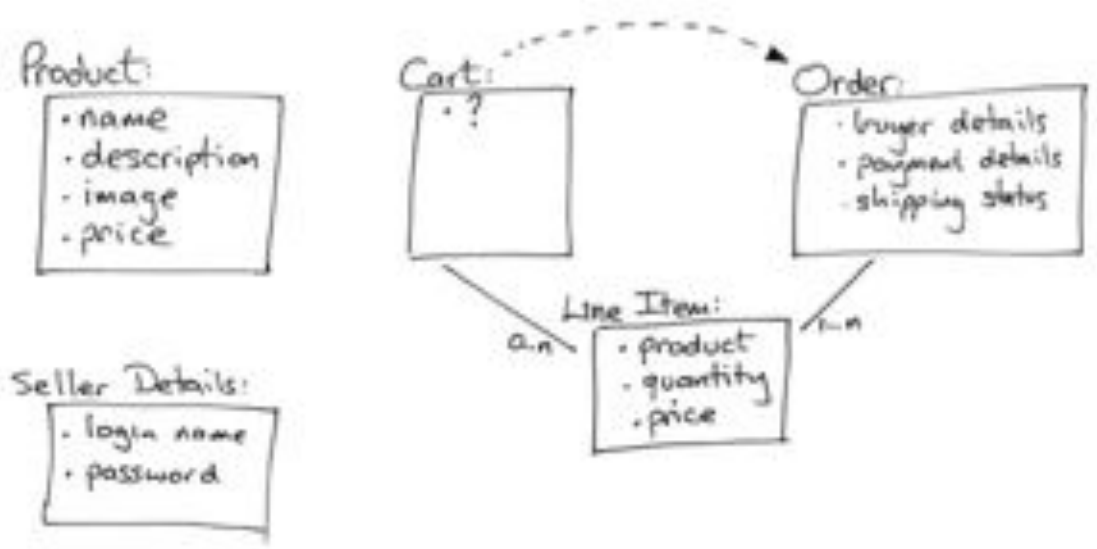


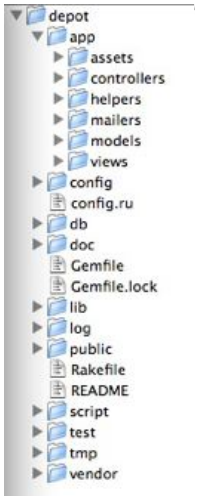
Figure 5.3: Initial guess at application data

Creating the application

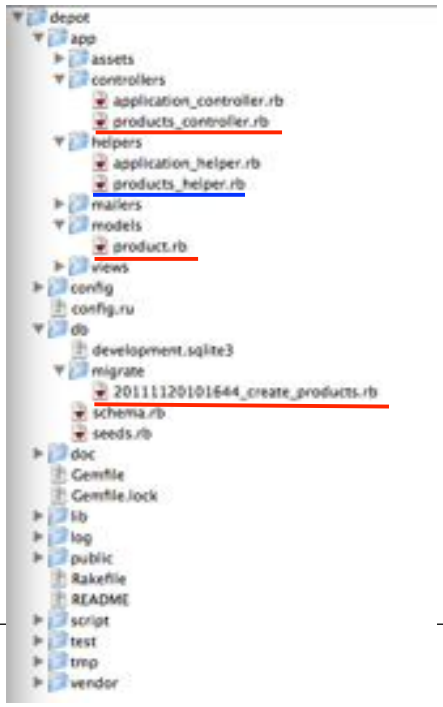
- Creating the Products Maintenance Application
  - creating a rails applications `rails new depot`
  - creating the Database
  - generating the Scaffold
    ```
    rails generate scaffold Product title:string
      description:text image_url:string price:decimal
    ```
  - applying the Migration `rake db:migrate`
  - seeing the List of Products `rails server`
  - adding products
  - adding test data `rake db:seed`
  - improving the default view of list of products
  -

Creating the application and generating the Scaffold

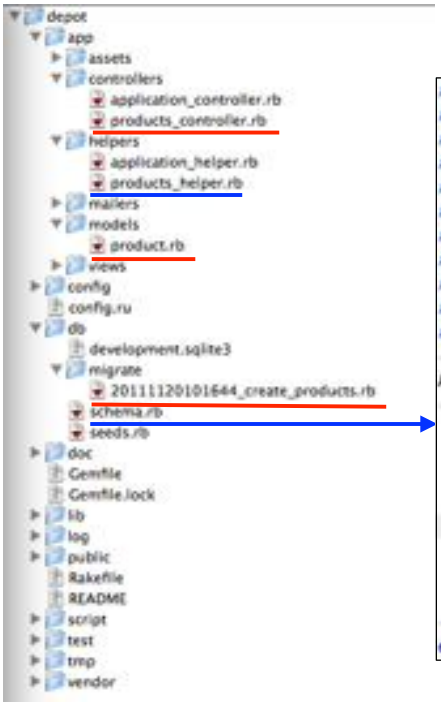
```
rails new depot
```




```
rails generate scaffold Product title:string
  description:text image_url:string price:decimal
```



Generating the Scaffold



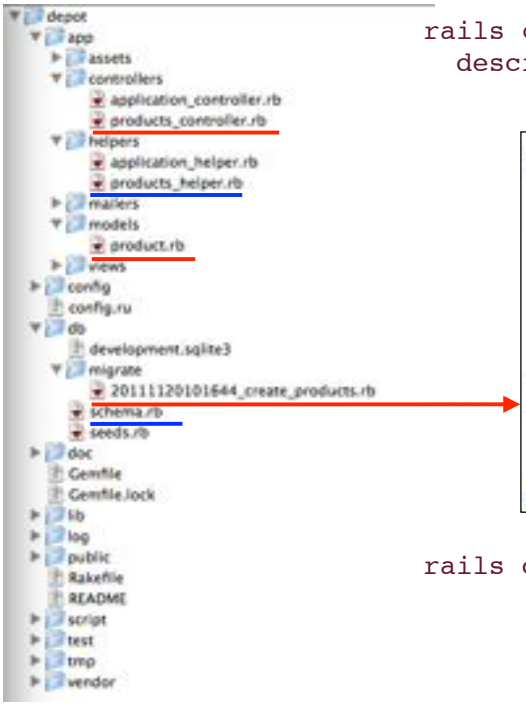

```
# This file is auto-generated from the current state of the database. Instead
# of editing this file, please use the migrations feature of Active Record to
# incrementally modify your database, and then regenerate this schema definitio
#
# Note that this schema.rb definition is the authoritative source for your
# database schema. If you need to create the application database on another
# system, you should be using db:schema:load, not running all the migrations
# from scratch. The latter is a flawed and unsustainable approach (the more mig
# you'll amass, the slower it'll run and the greater likelihood for issues).
#
# It's strongly recommended to check this file into your version control system.

ActiveRecord::Schema.define(:version => 20111120101644) do

  create_table "products", :force => true do |t|
    t.string   "title"
    t.text     "description"
    t.string   "image_url"
    t.decimal  "price"
    t.datetime "created_at"
    t.datetime "updated_at"
  end

end
```

Generating the Scaffold


```
rails generate scaffold Product title:string
  description:text image_url:string price:decimal
```

```
class CreateProducts < ActiveRecord::Migration
  def change
    create_table :products do |t|
      t.string :title
      t.text :description
      t.string :image_url
      t.decimal :price

      t.timestamps
    end
  end
end
```


Generating the Scaffold

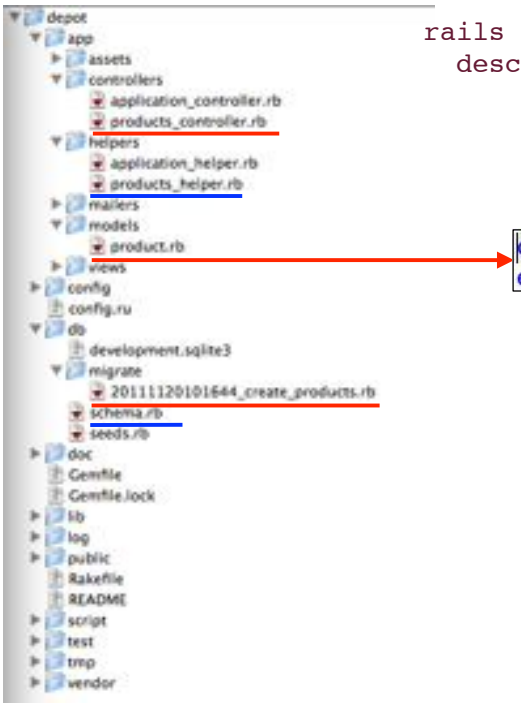



```
rails generate scaffold Product title:string
  description:text image_url:string price:decimal
```

```
class CreateProducts < ActiveRecord::Migration
  def change
    create_table :products do |t|
      t.string :title
      t.text :description
      t.string :image_url
      t.decimal :price, :precision => 8, :scale => 2

      t.timestamps
    end
  end
end
```

```
rails db:migrate
```

Generating the Scaffold



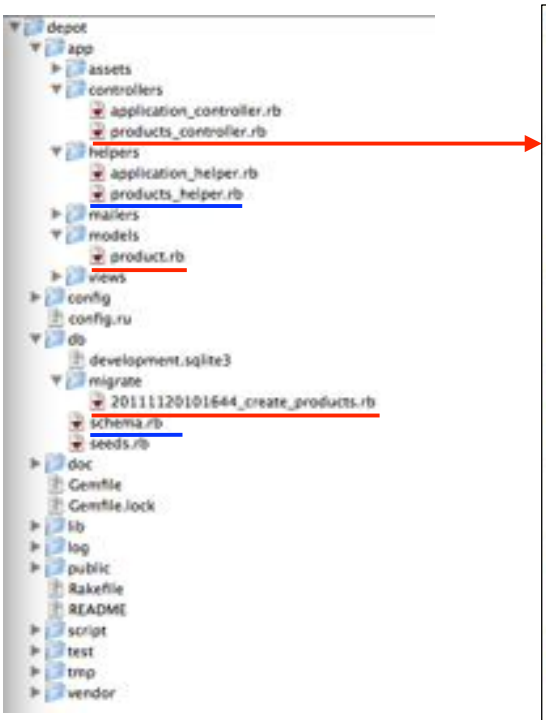

```
rails generate scaffold Product title:string
  description:text image_url:string price:decimal
```

```
class Product < ActiveRecord::Base
end
```


Generating the Scaffold

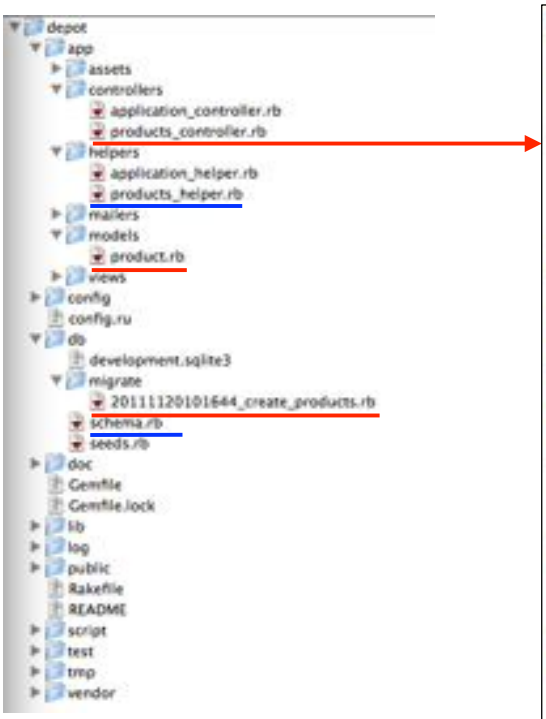

```
class ProductsController < ApplicationController
  # GET /products
  # GET /products.json
  def index
    @products = Product.all

    respond_to do |format|
      format.html # index.html.erb
      format.json { render :json => @products }
    end
  end

  # GET /products/1
  # GET /products/1.json
  def show

  # GET /products/new
  # GET /products/new.json
  def new

  # GET /products/1/edit
  def edit

  # POST /products
  # POST /products.json
  def create

  # PUT /products/1
  # PUT /products/1.json
  def update

  # DELETE /products/1
  # DELETE /products/1.json
  def destroy
end
```

Generating the Scaffold



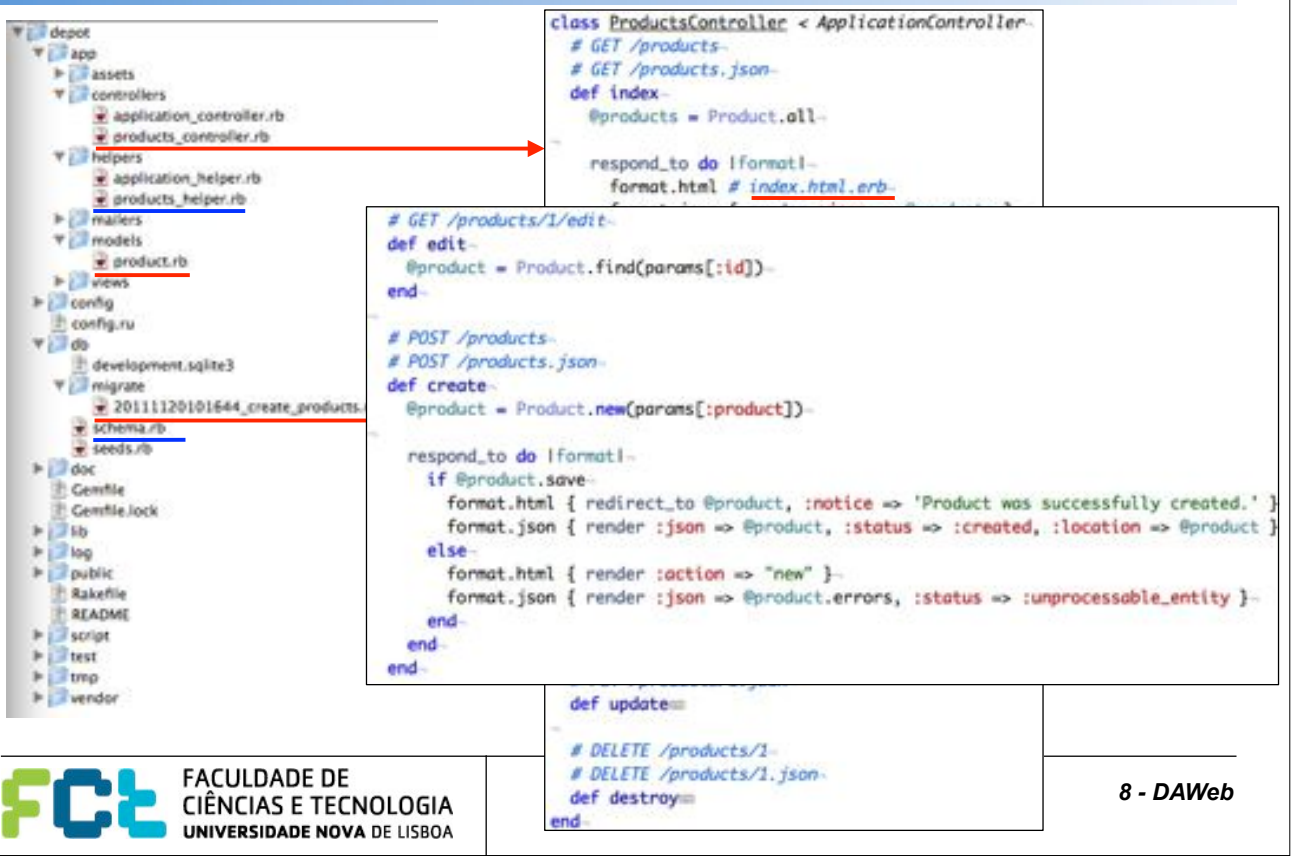

```ruby
class ProductsController < ApplicationController
  # GET /products
  # GET /products.json
  def index
    @products = Product.all

    respond_to do |format|
      format.html # index.html.erb
      format.json { render :json => @products }
    end
  end

  # GET /products/1
  # GET /products/1.json
  def show
    @product = Product.find(params[:id])

    respond_to do |format|
      format.html # show.html.erb
      format.json { render :json => @product }
    end
  end

  # GET /products/n
  # GET /products/n
  def new

  # GET /products/1
  def edit

  # POST /products
  # POST /products.json
  def create

  # PUT /products/1
  # PUT /products/1.json
  def update

  # DELETE /products/1
  # DELETE /products/1.json
  def destroy
end
```


Generating the Scaffold



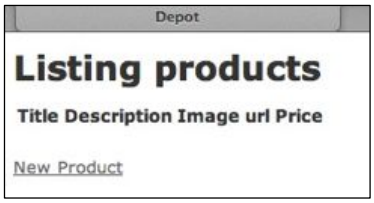

```ruby
class ProductsController < ApplicationController
  # GET /products
  # GET /products.json
  def index
    @products = Product.all

    respond_to do |format|
      format.html # index.html.erb
      format.json { render :json => @products }
    end
  end

  # GET /products/1
  # GET /products/1.json
  def show

  # GET /products/new
  # GET /products/new.json
  def new
    @product = Product.new

    respond_to do |format|
      format.html # new.html.erb
      format.json { render :json => @product }
    end
  end

  # GET /products/1/e
  def edit

  # POST /products
  # POST /products.js
  def create

  # PUT /products/1
  # PUT /products/1.json
  def update

  # DELETE /products/1
  # DELETE /products/1.json
  def destroy
end
```

Generating the Scaffold



rails server - http://localhost:3000/products

rails server - http://localhost:3000/products



products_controller.rb

```ruby
class ProductsController < ApplicationController
  # GET /products
  # GET /products.json
  def index
    @products = Product.all

    respond_to do |format|
      format.html # index.html.erb
      format.json { render :json => @products }
    end
```

index.html.erb

```erb
<h1>Listing products</h1>

<table>
  <tr>

<% @products.each do |product| %>
  <tr>
    <td><%= product.title %></td>
    <td><%= product.description %></td>
    <td><%= product.image_url %></td>
    <td><%= product.price %></td>
    <td><%= link_to 'Show', product %></td>
    <td><%= link_to 'Edit', edit_product_path(product) %></td>
    <td><%= link_to 'Destroy', product,
            :confirm => 'Are you sure?', :method => :delete %></td>
  </tr>
<% end %>
</table>

<br />

<%= link_to 'New Product', new_product_path %>
```


rails server - http://localhost:3000/products/new

products_controller.rb

```ruby
class ProductsController < ApplicationController

  ...
  # GET /products/new
  # GET /products/new.json
  def new
    @product = Product.new

    respond_to do |format|
      format.html # new.html.erb
      format.json { render :json => @product }
    end
  end
```

new.html.erb

```erb
<h1>New product</h1>

<%= render 'form' %>

<%= link_to 'Back', products_path %>
```

_form.html.erb

rails server - http://localhost:3000/products/new

_form.html.erb

```
<%= form_for(@product) do |f| %>
  <% if @product.errors.any? %>
    <div id="error_explanation">
  <% end %>

  <div class="field">
    <%= f.label :title %><br />
    <%= f.text_field :title %>
  </div>
  <div class="field">
    <%= f.label :description %><br />
    <%= f.text_area :description %>
  </div>
  <div class="field">
    <%= f.label :image_url %><br />
    <%= f.text_field :image_url %>
  </div>
  <div class="field">
    <%= f.label :price %><br />
    <%= f.text_field :price %>
  </div>
  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>
```


rails server - http://localhost:3000/new

_form.html.erb

```
<%= form_for(@product) do |f| %>
  <% if @product.errors.any? %>
    <div id="error_explanation">
  <% end %>

  <div class="field">
    <%= f.label :title %><br />
    <%= f.text_field :title %>
  </div>
  <div class="field">
    <%= f.label :description %><br />
    <%= f.text_area :description, :rows => 6 %>
  </div>
```

rails server - http://localhost:3000/products/ POST

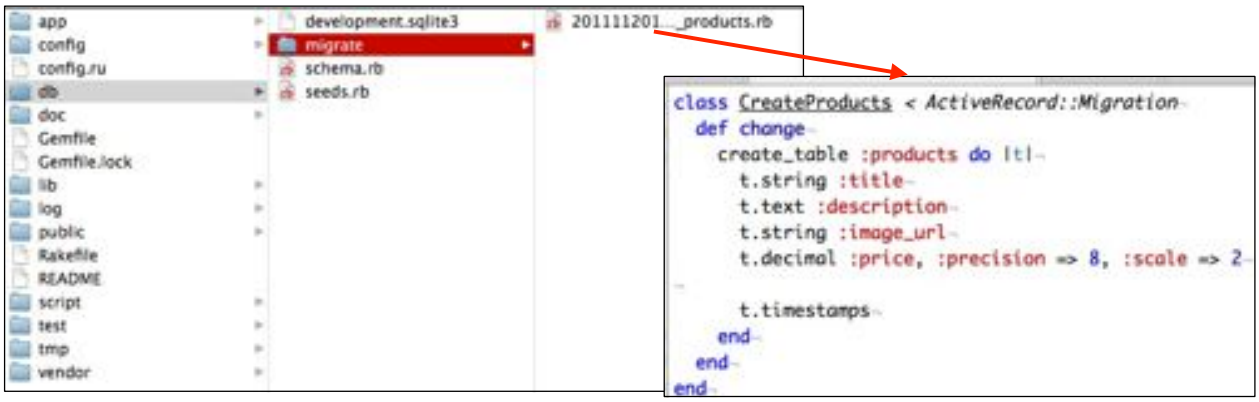
products_controller.rb

```
class ProductsController < ApplicationController
  # GET /products
  # GET /products.json
  def index

  # POST /products
  # POST /products.json
  def create
    @product = Product.new(params[:product])

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


Creating the application

- Creating the Products Maintenance Application
  - creating a rails applications `rails new depot`
  - creating the Database
  - generating the Scaffold
    ```
    rails generate scaffold Product title:string
      description:text image_url:string price:decimal
    ```
  - applying the Migration `rake db:migrate`
  - seeing the List of Products `rails server`
  - adding products
  - adding test data `rake db:seed`
  - improving the default view of list of products
  - ...

Adding test data



Adding test data


```
depot> rake db:seed
```

Adding test data

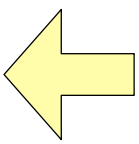
Listing products

Title	Description	Image url	Price			
Web Design for Developers	Web Design for Developers will show you how to make your web-based application look professionally designed. We'll help you learn how to pick the right colors and fonts, avoid costly interface and accessibility mistakes -- your application will really come alive. We'll also walk you through some common Photoshop and CSS techniques and work through a web site redesign, taking a new design from concept all the way to implementation.	/images/wd4d.jpg	42.95	Show	Edit	Destroy
Programming Ruby 1.9	Ruby is the fastest growing and most exciting dynamic language out there. If you need to get working programs delivered fast, you should add Ruby to your toolbox.	/images/ruby.jpg	49.5	Show	Edit	Destroy
Rails Test Prescriptions	Rails Test Prescriptions is a comprehensive guide to testing Rails applications, covering Test-Driven Development from both a theoretical perspective (why to test) and from a practical perspective (how to test effectively). It covers the core Rails testing tools and procedures for Rails 2 and Rails 3, and introduces popular add-ons, including Cucumber, Shoulda, Machinist, Mocha, and Rcov.	/images/rtp.jpg	43.75	Show	Edit	Destroy

New Product

Improving the default view of list of products

- Creating the Products Maintenance Application
  - creating a rails applications `rails new depot`
  - creating the Database
  - generating the Scaffold
    ```
    rails generate scaffold Product title:string
      description:text image_url:string price:decimal
    ```
  - applying the Migration `rake db:migrate`
  - seeing the List of Products `rails server`
  - adding products
  - adding test data `rake db:seed`
  - improving the default view of list of products
  - ...

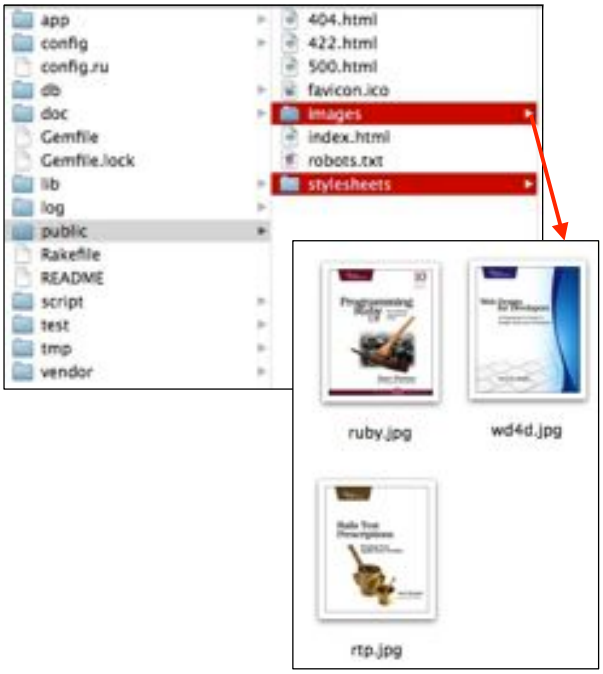


Improving the default view of list of products

- CSS
- Images
- Modifying the template

Improving the default view of list of products

- Images
- CSS
  -
  - Classes
    - list_line_odd, list_line_even
    - list_description
    - list_actions



Improving the default view of list of products

```
<div id="product_list">
    <h1>Listing products</h1>

    <table>

    <% @products.each do |product| %>
      <tr class="<%= cycle('list_line_odd', 'list_line_even') %>">
        <td>
          <%= image_tag(product.image_url, :class => 'list_image') %>
        </td>
        <td class="list_description">
          <dl>
            <dt><%= product.title %></dt>
            <dd><%= truncate(strip_tags(product.description),
                :length => 80) %></dd>
          </dl>
        </td>

        <td class="list_actions">
          <%= link_to 'Show', product %><br/>
          <%= link_to 'Edit', edit_product_path(product) %><br/>
          <%= link_to 'Destroy', product, :confirm => 'Are you sure?',
                                          :method => :delete %>
        </td>
      </tr>
    <% end %>
    </table>
</div>
<br />

<%= link_to 'New Product', new_product_path %>
```

removed the table headers

- definition: dl, dt, dd
- use of helper tuncate and strip_tags

http://localhost:3000/products/id,
GET

http://localhost:3000/products/id/edit
GET

http://localhost:3000/products/id
DELETE

Managing your development process

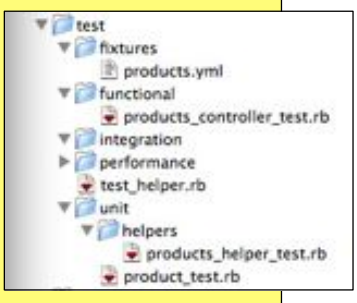
- Data and DataBase
  - `rake db:rollback`
  - `rake db:migrate`
  - `rake db:seed`
- Version Control
  - GIT

Testing

rake test

is for the **unit**, **functional**, and **integration** tests that Rails generates along with the scaffolding.

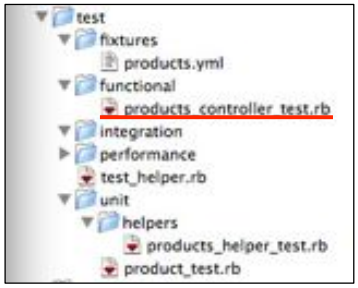
```
Joao-Moura-Pires-MacBook-Pro:depot joaomp$ rake test
Loaded suite /Library/Ruby/Gems/1.8/gems/rake-0.9.2.2/lib/rake/rake_test_loader
Started

Finished in 0.000163 seconds.

0 tests, 0 assertions, 0 failures, 0 errors
Loaded suite /Library/Ruby/Gems/1.8/gems/rake-0.9.2.2/lib/rake/rake_test_loader
Started
F.....F
Finished in 0.349474 seconds.

  1) Failure:
test_should_create_product(ProductsControllerTest) [test/functional/
products_controller_test.rb:20]:
"Product.count" didn't change by 1.
<3> expected but was
<2>.

  2) Failure:
test_should_update_product(ProductsControllerTest) [test/functional/
products_controller_test.rb:39]:
Expected response to be a <:redirect>, but was <200>.

7 tests, 9 assertions, 2 failures, 0 errors
Errors running test:functionals!
```



rake test

is for the **unit**, **functional**, and **integration** tests that Rails generates along with the scaffolding.

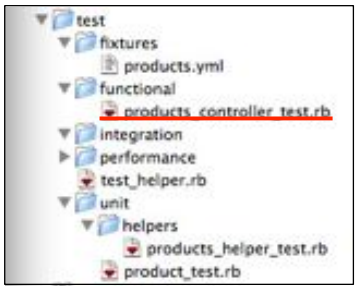
```
require 'test_helper'

class ProductsControllerTest < ActionController::TestCase
  setup do
    @product = products(:one)
    @update = {
      :title       => 'Lorem Ipsum',
      :description => 'Wibbles are fun!',
      :image_url   => 'lorem.jpg',
      :price       => 19.95
    }
  end

  test "should get index" do
    get :index
    assert_response :success
    assert_not_nil assigns(:products)
  end
```



rake test

is for the **unit**, **functional**, and **integration** tests that Rails generates along with the scaffolding.

```
  test "should get new" do
    get :new
    assert_response :success
  end

  test "should create product" do
    assert_difference('Product.count') do
      post :create, :product => @update
    end

    assert_redirected_to product_path(assigns(:product))
  end

  # ...
  test "should update product" do
    put :update, :id => @product.to_param, :product => @update
    assert_redirected_to product_path(assigns(:product))
  end

  # ...
end
```



Fixtures

- Fixtures are a **way of organizing data** that you want **to test against**; in short, sample data.
- They are stored in **YAML** files, **one file per model**, which are placed in the directory appointed by ActiveSupport::TestCase.fixture_path=(path) (this is automatically configured for Rails, so you can just put your files in <your-rails-app>/test/fixtures/).
- The fixture file ends with the **.yml file extension** (Rails example: <your-rails-app>/test/fixtures/web_sites.yml). The format of a fixture file looks like this:

```
rubyonrails:
  id: 1
  name: Ruby on Rails
  url: http://www.rubyonrails.org

google:
  id: 2
  name: Google
  url: http://www.google.com
```

products.yml

```
one:
  title: MyString
  description: MyText
  image_url: MyString
  price: 9.99

two:
  title: MyString
  description: MyText
  image_url: MyString
  price: 9.99
```


Fixtures

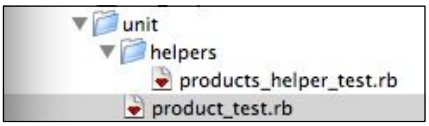
- Read:
  - http://api.rubyonrails.org/classes/ActiveRecord/Fixtures.html
- Other topics
  - Ordered fixtures, use the omap YAML type
  - Dynamic fixtures with ERB
  - Transactional Fixtures
  - Advanced Fixtures

Unit Testing of Models

- Scafolding

products_test.rb

```
require 'test_helper'

class ProductTest < ActiveSupport::TestCase
  # test "the truth" do
  #   assert true
  # end
end
```



products_helper_test.rb

```
require 'test_helper'

class ProductsHelperTest < ActionView::TestCase
end
```

- Rails generates tests based on the Test::Unit framework that comes preinstalled with Ruby.
  - An **assertion** is simply a method call that **tells the framework what we expect to be true**.
  - The simplest assertion is the method **`assert`**, which **expects its argument to be true**.
  - If it is, nothing special happens. However, **if the argument to assert is false**, the **assertion fails**. The framework will output a message and will stop executing the test method containing the failure.

Unit Testing of Models

- We expect that an **empty Product model will not pass validation**, so we can express that expectation by asserting that it isn't valid.

```
assert product.invalid?
```

products_test.rb

```
require 'test_helper'

class ProductTest < ActiveSupport::TestCase
  test "product attributes must not be empty" do
    product = Product.new
    assert product.invalid?
    assert product.errors[:title].any?
    assert product.errors[:description].any?
    assert product.errors[:price].any?
    assert product.errors[:image_url].any?
end
```

rake test:units

```
depot> rake test:units
Loaded suite lib/rake/rake_test_loader
Started
..
Finished in 0.092314 seconds.
1 tests, 5 assertions, 0 failures, 0 errors
```

Unit Testing of Models

- **Validation of the price** works the way we expect:

```
test "product price must be positive" do
product = Product.new(:title => "My Book Title",
                      :description => "yyy",
                      :image_url   => "zzz.jpg")
  product.price = -1
  assert product.invalid?
  assert_equal "must be greater than or equal to 0.01",
    product.errors[:price].join('; ')

  product.price = 0
  assert product.invalid?
  assert_equal "must be greater than or equal to 0.01",
    product.errors[:price].join('; ')

  product.price = 1
  assert product.valid?
end
```


Unit Testing of Models

- Validating that the **image URL ends with one** of .gif, .jpg, or .png:

```
def new_product(image_url)
Product.new(:title => "My Book Title",
            :description => "yyy",
            :price       => 1,
            :image_url   => image_url)
end

test "image url" do
  ok = %w{ fred.gif fred.jpg fred.png FRED.JPG FRED.Jpg
           http://a.b.c/x/y/z/fred.gif }

  bad = %w{ fred.doc fred.gif/more fred.gif.more }

  ok.each do |name|
    assert new_product(name).valid?, "#{name} shouldn't be invalid"
  end

  bad.each do |name|
    assert new_product(name).invalid?, "#{name} shouldn't be valid"
  end
end
```

Unit Testing of Models: using fixtures

- Our model contains a validation that checks that all the product titles in the database are unique.
  - To test this one, we're going to need to store product data in the database.
  - Fixtures
    - **Each fixture file** contains the data for a **single model**. The name of the fixture file is significant; **the base name of the file must match the name of a database table**.
    - Rails already created this fixture file when we first created the model:

```
one:
  title: MyString
  description: MyText
  image_url: MyString
  price: 9.99

two:
  title: MyString
  description: MyText
  image_url: MyString
  price: 9.99
```

In the case of the Rails-generated fixture, the rows are named *one* and *two*.

you **must use spaces**, not tabs, at the start of each of the data lines, and **all the lines for a row must have the same indentation**.

Unit Testing of Models: using fixtures

products_test.rb

```
class ProductTest < ActiveSupport::TestCase
  fixtures :products  # by default all fixtures are loaded
                      # adding the fixtures directive means that the products table
                      # will be emptied out and then populated with the three rows
                      # defined in the fixture before each test method is run.
```

- Rails needs to use a test database. If you look in the `database.yml` file in the config directory, you'll notice Rails actually created a configuration for three separate databases:
  - `db/development.sqlite3` will be our development database.
  - `db/test.sqlite3` is a test database.
  - `db/production.sqlite3` is the production database. Our application will use this when we put it online.
- **Each test method gets a freshly initialized table** in the `test` database, loaded from the fixtures we provide.

Unit Testing of Models: using fixtures

- Our model contains a validation that checks that all the product titles in the database are unique

products_test.rb

```
test "product is not valid without a unique title" do
product = Product.new(:title => products(:ruby).title,
                      :description => "yyy",
                      :price       => 1,
                      :image_url   => "fred.gif")
  assert !product.save
  assert_equal "has already been taken", product.errors[:title].join('; ')
end
```

- The test assumes that the database already includes a row for the Ruby book. It gets the title of that existing row using this: `products(:ruby).title`

Catalog Display (buyer)

Buyer and Seller

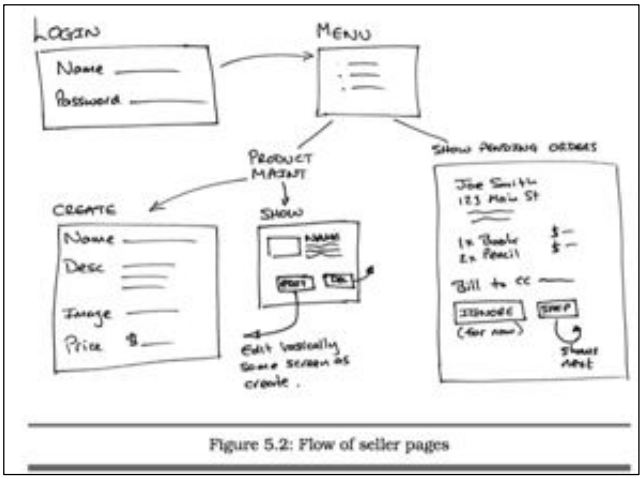


Figure 5.2: Flow of seller pages

Seller

`product_controller`

`store_controller`

Buyer

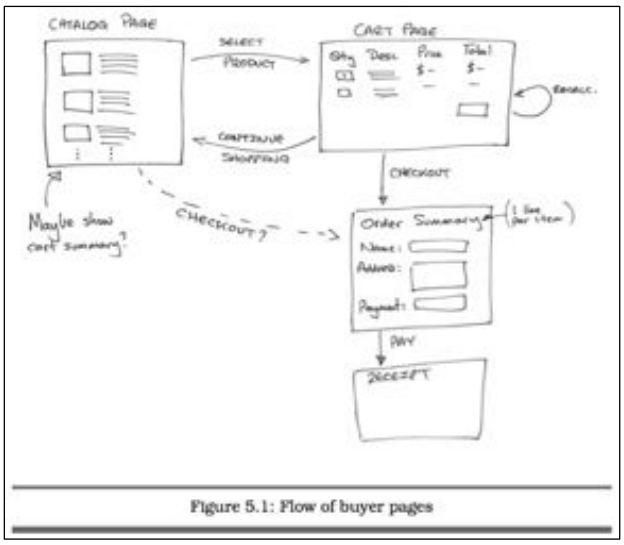


Figure 5.1: Flow of buyer pages

Creating the store controller

- Controller name: store
- Method: index

```
Joao-Moura-Pires-MacBook-Pro:depot joaoep$ rails generate controller store index
      create  app/controllers/store_controller.rb
       route  get "store/index"
      invoke  erb
      create    app/views/store
      create    app/views/store/index.html.erb
      invoke  test_unit
      create    test/functional/store_controller_test.rb
      invoke  helper
      create    app/helpers/store_helper.rb
      invoke    test_unit
      create      test/unit/helpers/store_helper_test.rb
      invoke  assets
      invoke    coffee
      create      app/assets/javascripts/store.js.coffee
      invoke    scss
      create      app/assets/stylesheets/store.css.scss
```

`http://localhost:3000/store/index`

Setting the root for the site

Depot

Store#index

Find me in app/views/store/index.html.erb

http://localhost:3000/store/index

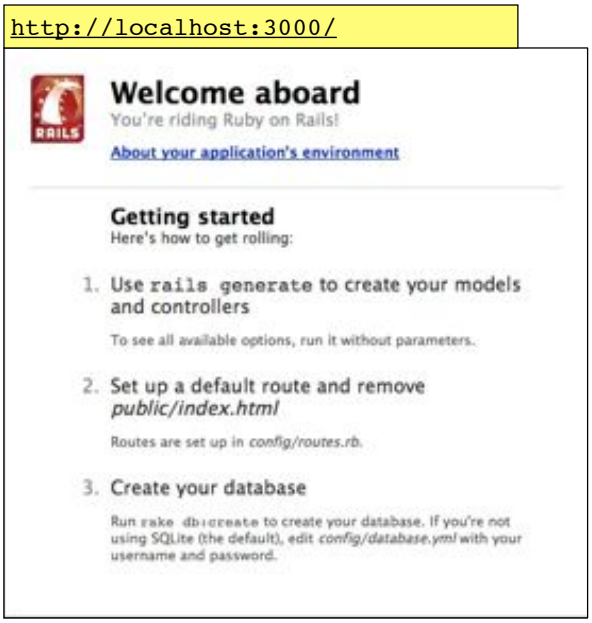
Action Controller: Exception caught

Routing Error

No route matches [GET] "/store"

http://localhost:3000/store

http://localhost:3000/



Setting the root for the site

Depot

Store#index

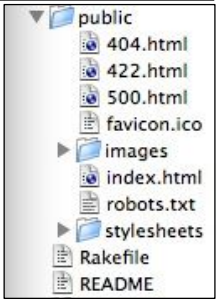
Find me in app/views/store/index.html.erb

http://localhost:3000/store/index

http://localhost:3000/

`rm public/index.html`

```
Depot::Application.routes.draw do
  get "store/index"

  resources :products

  # You can have the root of your site routed with "root"
  # just remember to delete public/index.html.
  # root :to => 'welcome#index'
  root :to => 'store#index', :as => 'store'

end
```



depot/config/routes.rb

Displaying a simple list of all the products

- Get the list of products out of the database and make it available to the code in the view that will display the table.

store_controller.rb

```
class StoreController < ApplicationController
  def index
    @products = Product.all
  end

end
```

index.html.erb

```
<% if notice %>
  <p id="notice"><%= notice %></p>
<% end %>

<h1>Your Pragmatic Catalog</h1>

<% @products.each do |product| %>
  <div class="entry">
    <%= image_tag(product.image_url) %>
    <h3><%= product.title %></h3>
    <%= sanitize(product.description) %>
    <div class="price_line">
      <span class="price"><%= product.price %></span>
    </div>
  </div>
<% end %>
```


Displaying a simple list of all the products

http://localhost:3000/

Displaying a simple list **ordered by title**

- **Default scopes apply to all queries that start with this model.**

product.rb

```
class Product < ActiveRecord::Base

  default_scope :order => 'title'

  validates :title, :description, :image_url, :presence => true
  validates :price, :numericality => {:greater_than_or_equal_to => 0.01}
  validates :title, :uniqueness => true
  validates :image_url, :format => {
     :with    => %r{\.(gif|jpg|png)$}i,
     :message => 'must be a URL for GIF, JPG or PNG image.'
  }
end
```


Displaying a simple list **ordered by title**

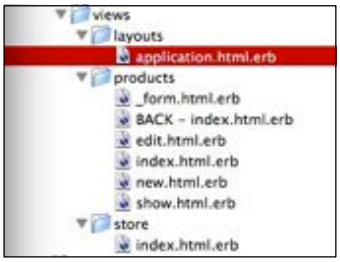
http://localhost:3000/

Adding a page Layout

- `application.html.erb` will be the layout used for all views for all controllers that don't otherwise provide a layout.
- By using only one layout, we can change the look and feel of the entire site by editing just one file.

`application.html.erb`

```
<!DOCTYPE html>
<html>
<head>
  <title>Depot</title>
  <%= stylesheet_link_tag    "application" %>
  <%= javascript_include_tag "application" %>
  <%= csrf_meta_tags %>
</head>
<body>

<%= yield %>

</body>
</html>
```



the content generated by the viewers

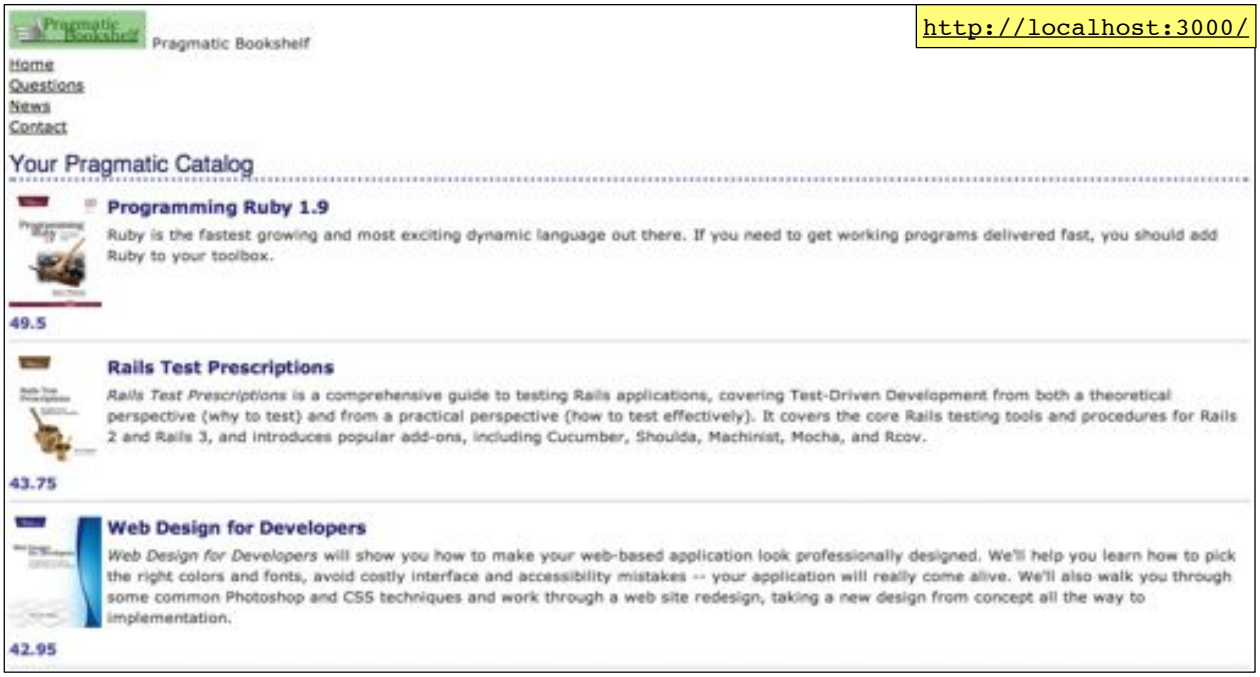
Adding a page Layout

`application.html.erb`

```
<!DOCTYPE html>
<html>
<head>
  <title>Pragprog Books Online Store</title>
  <%= stylesheet_link_tag "application" %>
  <%= stylesheet_link_tag "scaffold" %>
  <%= stylesheet_link_tag "depot", :media => "all" %>
  <%= javascript_include_tag "application" %>
  <%= csrf_meta_tags %>
</head>
<body id="store" >
    <div id="banner">
        <%= image_tag("logo.png") %>
        <%= @page_title || "Pragmatic Bookshelf" %>
    </div>
    <div id="columns">
        <div id="side">
            <a href="http://www...."">Home</a><br />
            <a href="http://www..../faq">Questions</a><br /> <a href="http://www..../news">News</a><br />
            <a href="http://www..../contact">Contact</a><br />
        </div>
        <div id="main">
            <%= yield %>
        </div>
    </div>
</body>
</html>
```

Adding a page Layout

http://localhost:3000/



Adding a page Layout

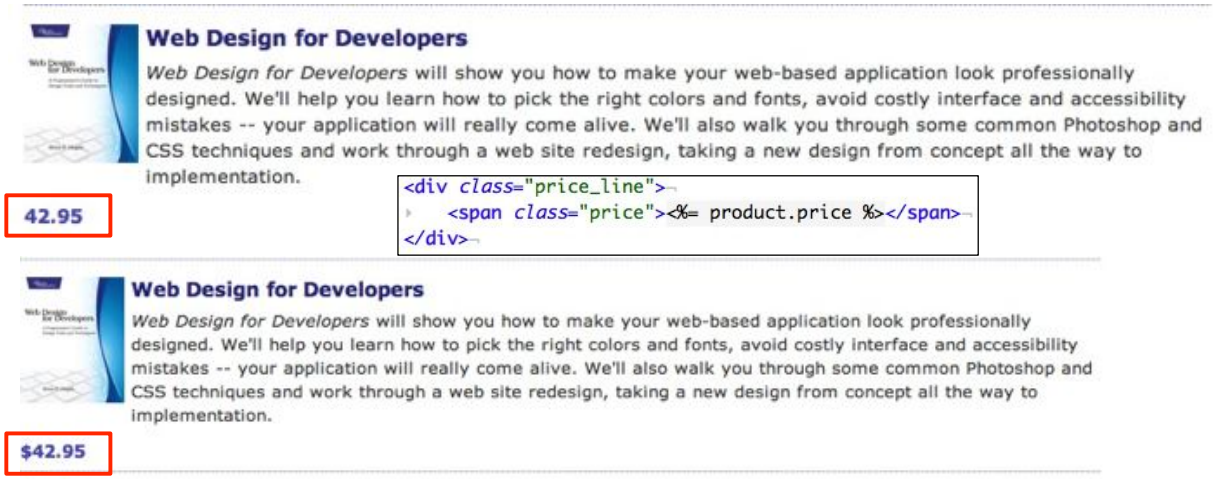
app/assets/stylesheets/depot.css

```
/* Styles for main page */
#banner {
  background: #9c9;
  padding-top: 10px;
  padding-bottom: 10px;
  border-bottom: 2px solid;
  font: small-caps 40px/40px "Times New Roman", serif;
  color: #282;
  text-align: center;
}
#banner img { float: left;
}
#columns {
  background: #141;
}

#main {
  margin-left: 17em;
  padding-top: 4ex;
  padding-left: 2em;
  background: white;
}
#side {
  float: left;
  padding-top: 1em;
  padding-left: 1em;
  padding-bottom: 1em;
  width: 16em;
  background: #141;
}
#side a {
color: #bfb; font-size: small;
}
```

Adding a page Layout

http://localhost:3000/

Using a Helper to Format the Price



- we can format the number by:
  - `<span class="price"><%= sprintf("$%0.02f", product.price) %></span>`
  - `<span class="price"><%= number_to_currency(product.price) %></span>`

Using a Helper to Format the Price

number_to_currency(number, options = {})

Formats a `number` into a currency string (e.g., $13.65). You can customize the format in the `options` hash.

Options

- `:locale` - Sets the locale to be used for formatting (defaults to current locale).
- `:precision` - Sets the level of precision (defaults to 2).
- `:unit` - Sets the denomination of the currency (defaults to "$").
- `:separator` - Sets the separator between the units (defaults to ".").
- `:delimiter` - Sets the thousands delimiter (defaults to ",").
- `:format` - Sets the format for non-negative numbers (defaults to "%u%n").

```
Fields are <tt>%u</tt> for the currency, and <tt>%n</tt>
for the number.
```

- `:negative_format` - Sets the format for negative numbers (defaults to prepending

```
an hyphen to the formatted number given by <tt>:format</tt>).
Accepts the same fields than <tt>:format</tt>, except
<tt>%n</tt> is here the absolute value of the number.
```


Using a Helper to Format the Price

Examples

```
number_to_currency(1234567890.50)                    # => $1,234,567,890.50
number_to_currency(1234567890.506)                   # => $1,234,567,890.51
number_to_currency(1234567890.506, :precision => 3)  # => $1,234,567,890.506
number_to_currency(1234567890.506, :locale => :fr)   # => 1 234 567 890,51 €

number_to_currency(-1234567890.50, :negative_format => "(%u%n)")
# => ($1,234,567,890.50)
number_to_currency(1234567890.50, :unit => "&pound;", :separator => ",", :delimiter => "")
# => &pound;1234567890,50
number_to_currency(1234567890.50, :unit => "&pound;", :separator => ",", :delimiter => "", :format => "%n %u")
# => 1234567890,50 &pound;
```

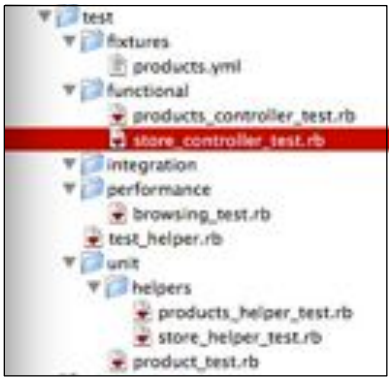
Functional Testing of Controllers

- The unit testing of models that we did previously seemed straightforward enough. We called a method and compared what it returned against what we expected it to return.
- But now we are dealing with a server that processes requests and a user viewing responses in a browser. What we will need is **functional tests that verify that the model, view, and controller work well together**.

```
depot> rake test:functionals
```

teste/functional/store_controller_test.rb

```
require 'test_helper'

class StoreControllerTest < ActionController::TestCase
  test "should get index" do
    get :index
    assert_response :success
  end

end
```



Functional Testing of Controllers

- We want also verify that the response contains our **layout**, our product **information**, and our **number formatting**.

teste/functional/store_controller_test.rb

```
require 'test_helper'

class StoreControllerTest < ActionController::TestCase
  test "should get index" do
    get :index
    assert_response :success
    assert_select '#columns #side a', :minimum => 4
    assert_select '#main .entry', 3
    assert_select 'h3', 'Programming Ruby 1.9'
    assert_select '.price', /\$[,\d]+\.\d\d/
  end

end
```

These assertions are based on the test data that we had put inside our fixtures:

- This test verifies that there are a minimum of four links inside an element with an ID #side, ... (layout).
- The next three lines verify that all of our products are correctly displayed.

Functional Testing of Controllers

assert_select(*args, &block)

An assertion that selects elements and makes one or more equality tests.

If the first argument is an element, selects all matching elements starting from (and including) that element and all its children in depth-first order.

If no element if specified, calling `assert_select` selects from the response HTML unless `assert_select` is called from within an `assert_select` block.

When called with a block `assert_select` passes an array of selected elements to the block. Calling `assert_select` from the block, with no element specified, runs the assertion on the complete set of elements selected by the enclosing assertion. Alternatively the array may be iterated through so that `assert_select` can be called separately for each element.

Example

If the response contains two ordered lists, each with four list elements then:

```
assert_select "ol" do |elements|
  elements.each do |element|
    assert_select element, "li", 4
  end
end
```

will pass, as will:

```
assert_select "ol" do
  assert_select "li", 8
end
```

The selector may be a CSS selector expression (String), an expression with substitution values, or an HTML::Selector object.

Functional Testing of Controllers

Equality Tests

The equality test may be one of the following:

- `true` - Assertion is true if at least one element selected.
- `false` - Assertion is true if no element selected.
- `String/Regexp` - Assertion is true if the text value of at least one element matches the string or regular expression.
- `Integer` - Assertion is true if exactly that number of elements are selected.
- `Range` - Assertion is true if the number of selected elements fit the range.

If no equality test specified, the assertion is true if at least one element selected.

To perform more than one equality tests, use a hash with the following keys:

- `:text` - Narrow the selection to elements that have this text value (string or regexp).
- `:html` - Narrow the selection to elements that have this HTML content (string or regexp).
- `:count` - Assertion is true if the number of selected elements is equal to this value.
- `:minimum` - Assertion is true if the number of selected elements is at least this value.
- `:maximum` - Assertion is true if the number of selected elements is at most this value.

Functional Testing of Controllers

Examples

```
# At least one form element
assert_select "form"

# Form element includes four input fields
assert_select "form input", 4

# Page title is "Welcome"
assert_select "title", "Welcome"

# Page title is "Welcome" and there is only one title element
assert_select "title", {:count => 1, :text => "Welcome"},
    "Wrong title or more than one title element"

# Page contains no forms
assert_select "form", false, "This page must contain no forms"

# Test the content and style
assert_select "body div.header ul.menu"

# Use substitution values
assert_select "ol>li#?", /item-\d+/

# All input fields in the form have a name
assert_select "form input" do
  assert_select "[name=?]", /.+/  # Not empty
end
```


Cart Creation (buyer)

Cart Creation

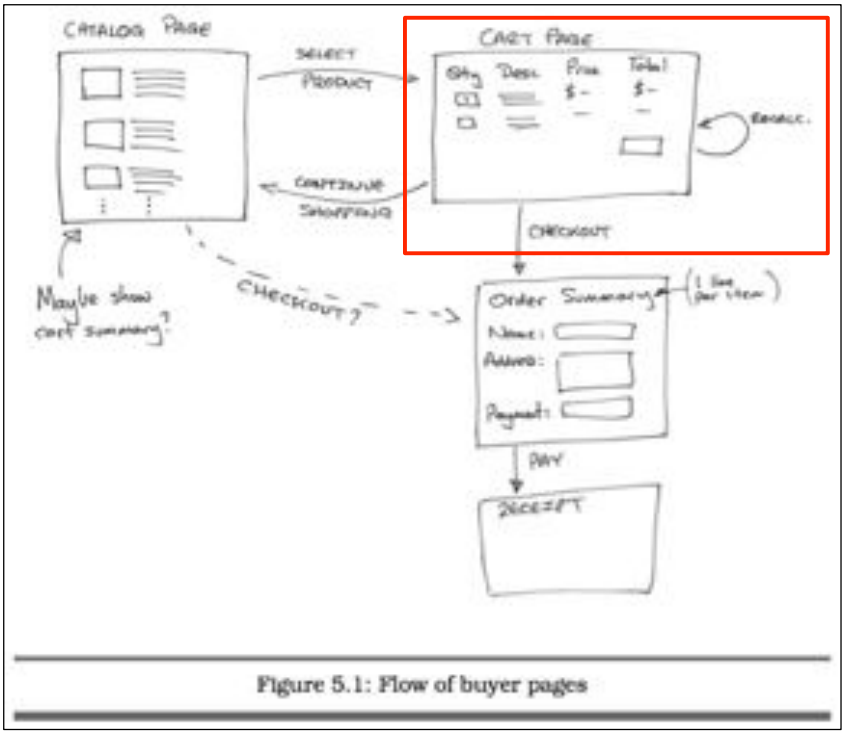


Figure 5.1: Flow of buyer pages

Cart creation

- Our application will need to **keep track** of **all the items added to the cart** by the **buyer**:
  - We'll keep a **cart in the database** and store its **unique identifier**, `cart.id`, in the **session**.
  - Every time a request comes in, we can recover the identity from the session and use it to find the cart in the database.

```
depot> rails generate scaffold cart
```

```
depot> rake db:migrate
```

Cart creation

```
ActiveRecord::Schema.define(:version => 20111126163937) do

  create_table "carts", :force => true do |t|
    t.datetime "created_at"
    t.datetime "updated_at"
  end

  create_table "products", :force => true do |t|
    t.string   "title"
    t.text     "description"
    t.string   "image_url"
    t.decimal  "price",       :precision => 8, :scale => 2
    t.datetime "created_at"
    t.datetime "updated_at"
  end

end
```

db/schema.rb

Cart creation - Getting from the Session

- We'll keep a **cart in the database** and store its **unique identifier**, `cart.id`, in the **session**.
- Every time a request comes in, we can recover the identity from the session and use it to find the cart in the database.

- **Rails makes the current session look like a hash to the controller**,
  - so we'll **store the id of the cart** in the **session by indexing it with the symbol** `:cart_id`.

Cart creation - Getting from the Session

- **Rails makes the current session look like a hash to the controller**,
  - so we'll **store the id of the cart** in the **session by indexing it with the symbol** `:cart_id`.

db/application_controller.rb

```
class ApplicationController < ActionController::Base
  protect_from_forgery

  private

    def current_cart
      Cart.find(session[:cart_id])
    rescue ActiveRecord::RecordNotFound
      cart = Cart.create
      session[:cart_id] = cart.id
      cart
    end
end
```

- The current_cart starts by getting the `:cart_id` from the session object and then attempts to find a cart corresponding to this `id`.
- If such a cart record is not found, then a new Cart is created, store the `id` of the created cart into the session, and then return the new cart.

- Being **`current_chart`** a **private** method of **`ApplicationController`** means that this method **available only to controllers** prevents Rails from making it available as an action on the controller.

Cart creation - Connecting Products to Carts

- **A cart contains a set of products.**

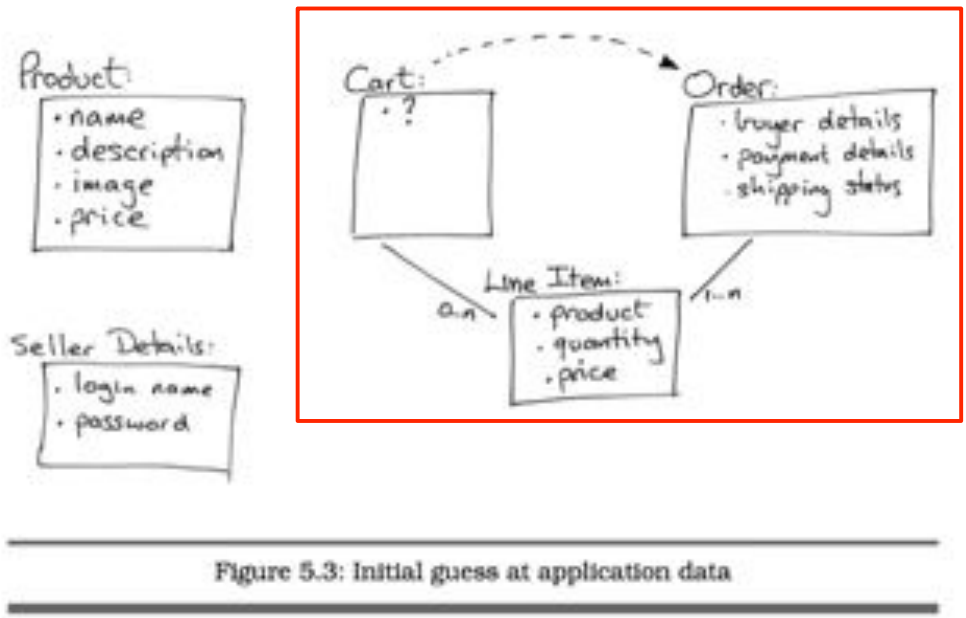


Figure 5.3: Initial guess at application data

Cart creation - Connecting Products to Carts

- Generate the Rails models and populate the migrations to create the corre- sponding tables:

```
depot> rails generate scaffold line_item product_id:integer  cart_id:integer
```

```
depot> rake db:migrate
```


```
class LineItem < ActiveRecord::Base
end
```

models/line_item.rb

Cart creation - Connecting Products to Carts

- Generate the Rails models and populate the migrations to create the corresponding tables:

```
depot> rake db:migrate
```

```
class LineItem < ActiveRecord::Base
end
```

models/line_item.rb

```
class Cart < ActiveRecord::Base
end
```

models/cart.rb

```
ActiveRecord::Schema.define(:version => 20111126173955) do

  create_table "carts", :force => true do |t|
    t.datetime "created_at"
    t.datetime "updated_at"
  end

  create_table "line_items", :force => true do |t|
    t.integer  "product_id"
    t.integer  "cart_id"
    t.datetime "created_at"
    t.datetime "updated_at"
  end

  create_table "products", :force => true do |t|
    t.string   "title"
    t.text     "description"
    t.string   "image_url"
    t.decimal  "price",       :precision => 8, :scale => 2
    t.datetime "created_at"
    t.datetime "updated_at"
  end

end
```

db/schema.rb

FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE NOVA DE LISBOA

8 - DAWeb

Cart creation - Connecting Products to Carts

```ruby
class LineItem < ActiveRecord::Base
  belongs_to :product
  belongs_to :cart
end
```

models/line_item.rb

- if a table has **foreign keys**, the corresponding model should have a **belongs_to** for each.

```ruby
class Cart < ActiveRecord::Base
  has_many :line_items, :dependent => :destroy
end
```

models/cart.rb

Cart creation - Connecting Products to Carts

- Navigation capabilities of the model objects.

```ruby
class LineItem < ActiveRecord::Base
  belongs_to :product
  belongs_to :cart
end
```

models/line_item.rb

```ruby
li = LineItem.find(...)
puts "This line item is for #{li.product.title}"
```

```ruby
class Cart < ActiveRecord::Base
  has_many :line_items, :dependent => :destroy
end
```

models/cart.rb

```ruby
cart = Cart.find(...)
puts "This cart has #{cart.line_items.count} line items"
```

Cart creation - Connecting Products to Carts

- Add a `has_many` directive to our Product

```ruby
class Product < ActiveRecord::Base

  default_scope :order => 'title'
  has_many :line_items

  before_destroy :ensure_not_referenced_by_any_line_item

  private
    # ensure that there are no line items referencing this product
    def ensure_not_referenced_by_any_line_item
      if line_items.empty?
        return true
      else
        errors.add(:base, 'Line Items present')
        return false
      end
    end
```

models/product.rb

Cart creation - Adding a Button

- To add an **Add to Cart** button for each product.
  - Purpose:
    - To add a new line_item
    - based on the current cart
    - and on the selected product

method

- `link_to` links to using HTTP GET.
- **`button_to`** links to using the **HTTP POST**

URL

- URL: `line_items_path`.

Parameter

- Which product to add to our cart?
  - **`:product_id`** option to the **`line_items_path`**.

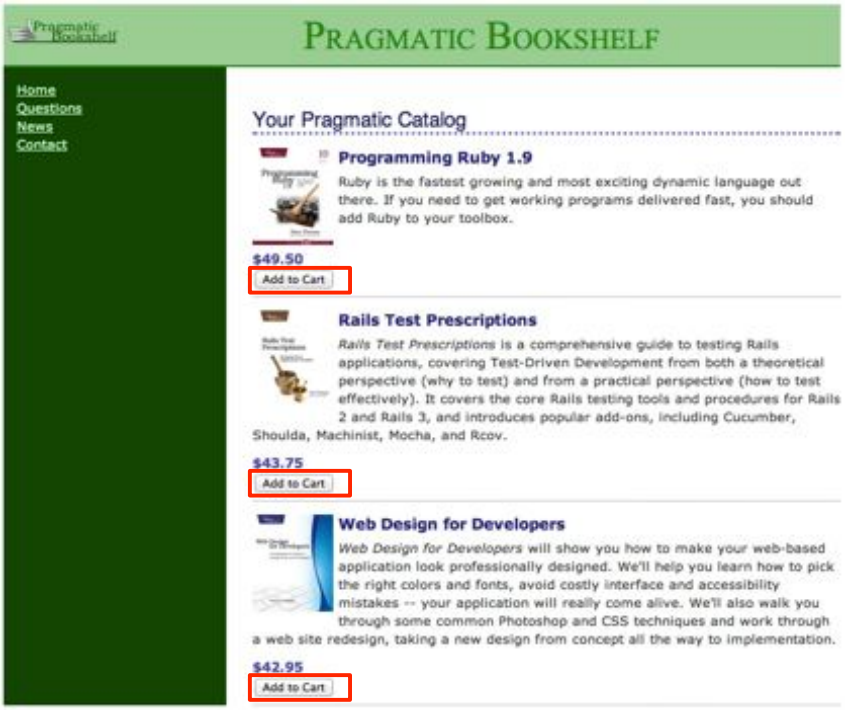
```ruby
class LineItemsController < ApplicationController
  # GET /line_items
  # GET /line_items.json
  def index

  # GET /line_items/1
  # GET /line_items/1.json
  def show

  # GET /line_items/new
  # GET /line_items/new.json
  def new

  # GET /line_items/1/edit
  def edit

  # POST /line_items
  # POST /line_items.json
  def create

  # PUT /line_items/1
  # PUT /line_items/1.json
  def update

  # DELETE /line_items/1
  # DELETE /line_items/1.json
  def destroy
end
```

Cart creation - Adding a Button

views/store/index.html.erb

```
<% if notice %>
<p id="notice"><%= notice %></p>
<% end %>

<h1>Your Pragmatic Catalog</h1>

<% @products.each do |product| %>
  <div class="entry">
    <%= image_tag(product.image_url) %>
    <h3><%= product.title %></h3>
    <%= sanitize(product.description) %>
    <div class="price_line">
      <span class="price"><%= number_to_currency(product.price) %></span>
      <%= button_to 'Add to Cart', line_items_path(:product_id => product) %>
    </div>
  </div>
<% end %>
```


Cart creation - Adding a Button



FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE NOVA DE LISBOA

8 - DAWeb

Cart creation - Adding a Button

`button_to` creates an HTML `<form>`, and that form contains an HTML `<div>`.

```
#store .entry form, #store .entry form div {
  display: inline;
}
```

```
<form action="/line_items?product_id=2"
class="button_to" method="post">
```

Cart creation - Adding a Button

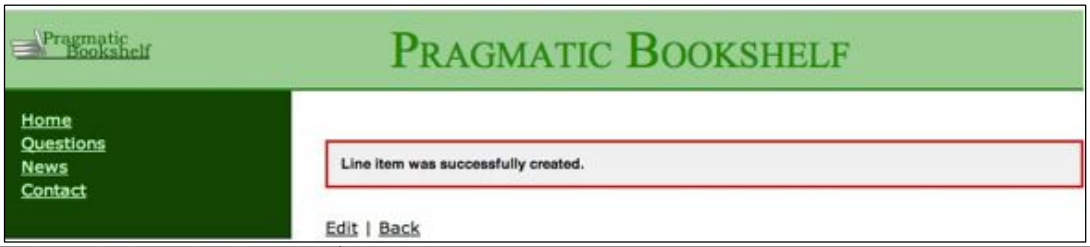
`line_items_controller.rb` **(scafold)**

```ruby
# POST /line_items
# POST /line_items.json
def create
  @line_item = LineItem.new(params[:line_item])

  respond_to do |format|
    if @line_item.save
      format.html { redirect_to @line_item, :notice => 'Line item was successfully created.' }
      format.json { render :json => @line_item, :status => :created, :location => @line_item }
    else
      format.html { render :action => "new" }
      format.json { render :json => @line_item.errors, :status => :unprocessable_entity }
    end
  end
end
```

Cart creation - Adding a Button

line_items_controller.rb **MODIFIED**

```ruby
# POST /line_items
# POST /line_items.json
def create
  @cart = current_cart
  product = Product.find(params[:product_id])
  @line_item = @cart.line_items.build(:product => product)

  respond_to do |format|
    if @line_item.save
      format.html { redirect_to @line_item.cart, :notice => 'Line item was successfully created.' }
      format.json { render :json => @line_item, :status => :created, :location => @line_item }
    else
      format.html { render :action => "new" }
      format.json { render :json => @line_item.errors, :status => :unprocessable_entity }
    end
  end
end
```



Cart creation - Adding a Button

line_items_controller.rb **MODIFIED**

```ruby
# POST /line_items
# POST /line_items.json
def create
  @cart = current_cart
  product = Product.find(params[:product_id])
  @line_item = @cart.line_items.build(:product => product)

  respond_to do |format|
    if @line_item.save
      format.html { redirect_to @line_item.cart, :notice => 'Line item was successfully created.' }
      format.json { render :json => @line_item, :status => :created, :location => @line_item }
    else
      format.html { render :action => "new" }
      format.json { render :json => @line_item.errors, :
    end
  end
end
```

current_cart method to find (or create) a cart in the session.

db/application_controller.rb

```ruby
class ApplicationController < ActionController::Base
  protect_from_forgery

  private

    def current_cart
      Cart.find(session[:cart_id])
    rescue ActiveRecord::RecordNotFound
      cart = Cart.create
      session[:cart_id] = cart.id
      cart
    end
end
```

FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE NOVA DE LISBOA

8 - DAWeb

Cart creation - Adding a Button

line_items_controller.rb **MODIFIED**

```ruby
# POST /line_items
# POST /line_items.json
def create
  @cart = current_cart
  product = Product.find(params[:product_id])
  @line_item = @cart.line_items.build(:product => product)

  respond_to do |format|
    if @line_item.save
      format.html { redirect_to @line_item.cart, :notice => 'Line item was successfully created.' }
      format.json { render :json => @line_item, :status => :created, :location => @line_item }
    else
      format.html { render :action => "new" }
      format.json { render :json => @line_item.errors, :status => :unprocessable_entity }
    end
  end
end
```

use the params object to get the :product_id parameter from the request.

The **params object** is important inside Rails applications.
It holds all of the parameters passed in a browser request.

We store the result in a local variable because there is no need to make this available to the view.

Cart creation - Adding a Button

line_items_controller.rb **MODIFIED**

```ruby
# POST /line_items
# POST /line_items.json
def create
  @cart = current_cart
  product = Product.find(params[:product_id])
  @line_item = @cart.line_items.build(:product => product)

  respond_to do |format|
    if @line_item.save
      format.html { redirect_to @line_item.cart, :notice => 'Line item was successfully created.' }
      format.json { render :json => @line_item, :status => :created, :location => @line_item }
    else
      format.html { render :action => "new" }
      format.json { render :json => @line_item.errors, :status => :unprocessable_entity }
    end
  end
end
```

+ pass the found product into @cart.line_items.build.
+ new **line_item relationship** to be built between the **@cart** object and the **product.**
+ save the resulting line item into an instance variable named **@line_item.**

Cart creation - Adding a Button

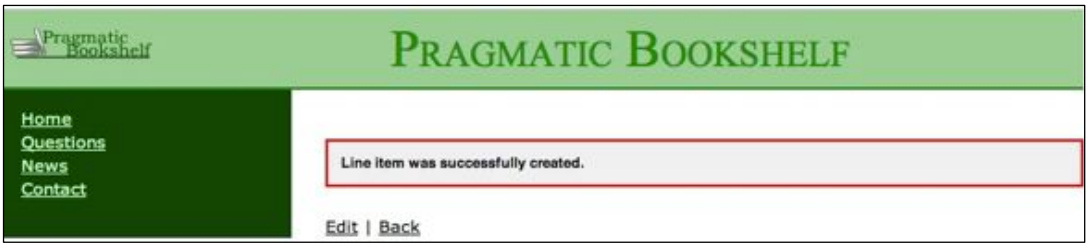
line_items_controller.rb **MODIFIED**

```
# POST /line_items
# POST /line_items.json
def create
  @cart = current_cart
  product = Product.find(params[:product_id])
  @line_item = @cart.line_items.build(:product => product)

  respond_to do |format|
    if @line_item.save
      format.html { redirect_to @line_item.cart, :notice => 'Line item was successfully created.' }
      format.json { render :json => @line_item, :status => :created, :location => @line_item }
    else
      format.html { render :action => "new" }
      format.json { render :json => @line_item.errors, :status => :unprocessable_entity }
    end
  end
end
```

we want to **redirect** you **to the cart** instead of back to the line item itself.

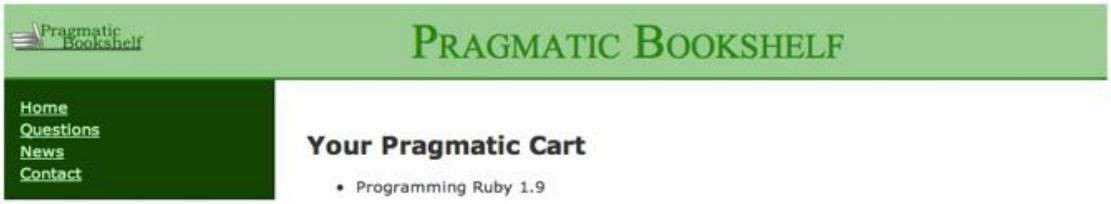
Since the line item object knows how to find the cart object, all we need to do is add .cart to the method call.

Cart creation - Adding a Button



views/carts/show.html.erb

```
<h2>Your Pragmatic Cart</h2>
<ul>
  <% @cart.line_items.each do |item| %>
    <li><%= item.product.title %></li>
  <% end %>
</ul>
```



Creating a Smarter Cart

- Associating a count with each product in our cart is going to require us to modify the line_items table.

```
depot> rails generate migration add_quantity_to_line_items quantity:integer
```

> The two patterns that Rails matches on is add_XXX_to_TABLE and remove_XXX_from_TABLE where the value of XXX is ignored; what matters is the list of column names and types that appear after the migration name.

```
class AddQuantityToLineItems < ActiveRecord::Migration
  def change
    add_column :line_items, :quantity, :integer
  end
end
```

Creating a Smarter Cart

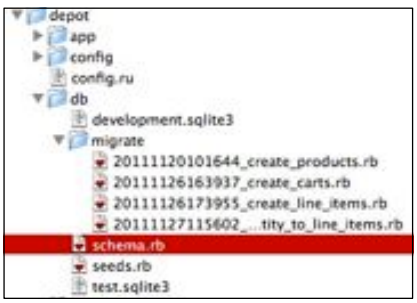
- Associating a count with each product in our cart is going to require us to modify the line_items table.

```
depot> rails generate migration add_quantity_to_line_items quantity:integer
```

```
class AddQuantityToLineItems < ActiveRecord::Migration
  def change
    add_column :line_items, :quantity, :integer, :default => 1
  end
end
```

```
depot> rake db:migrate
```

```
create_table "line_items", :force => true do |t|
  t.integer  "product_id"
  t.integer  "cart_id"
  t.datetime "created_at"
  t.datetime "updated_at"
  t.integer  "quantity",    :default => 1
end
```



Creating a Smarter Cart

- Now we need a smart **`add_product`** method in our Cart, one that checks whether our list of items already includes the product we're adding; if it does, it bumps the quantity, and if it doesn't, it builds a new LineItem:

models/cart.db

dynamic finder

```
class Cart < ActiveRecord::Base
  has_many :line_items, :dependent => :destroy

  def add_product(product_id)
    current_item = line_items.find_by_product_id(product_id)
    if current_item
      current_item.quantity += 1
    else
      current_item = line_items.build(:product_id => product_id)
    end
    current_item
  end

end
```

For **every field** (also known as an attribute) you define in your table, Active Record **provides a finder method.** If you have a field called first_name on your Client model for example, you get find_by_first_name and find_all_by_first_name for free from Active Record.

Creating a Smarter Cart

- We also need to modify the line item controller to make use of this method:

controllers/line_items_controller.db

```
def create
  @cart = current_cart
  product = Product.find(params[:product_id])
  @line_item = @cart.line_items.build(:product => product)

  respond_to do |format|
    if @line_item.save
      format.html { redirect_to @line_item.cart, :notice => 'Line item was successfully created.' }
      format.json { render :json => @line_item, :status => :created, :location => @line_item }
    else
      format.html { render :action => "new" }
      format.json { render :json => @line_item.errors, :status => :unprocessable_entity }
    end
  end
end
```

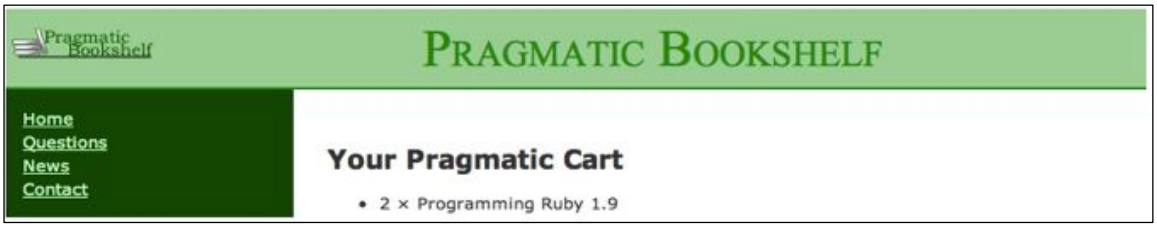
```
@line_item = @cart.add_product(product.id)
```

Creating a Smarter Cart

- How to view the quantity in the Cart:

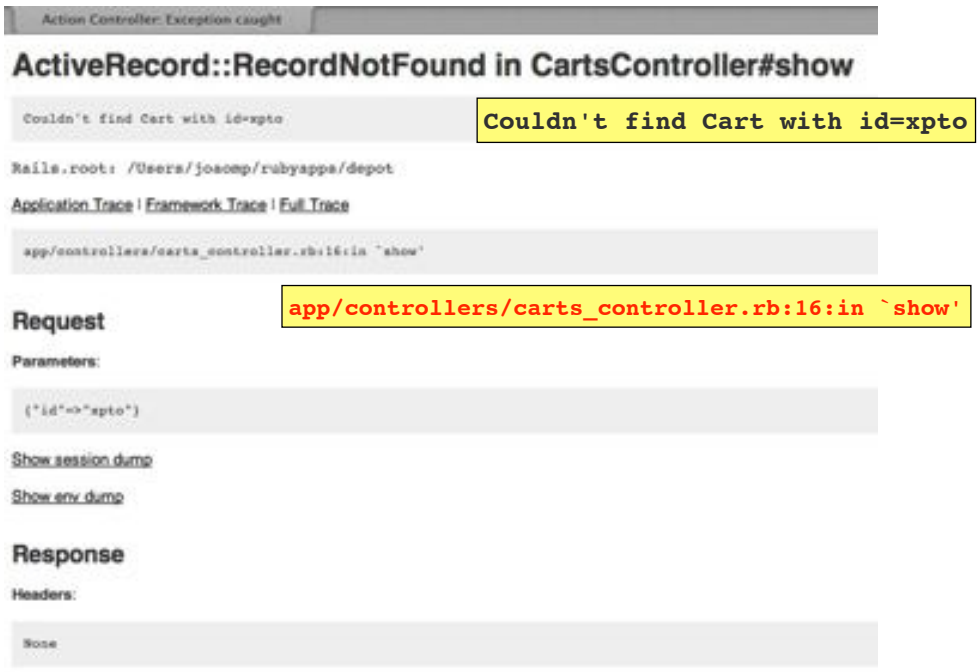
views/catrs/show.html.erb

```
<h2>Your Pragmatic Cart</h2>
<ul>
  <% @cart.line_items.each do |item| %>
    <li><%= item.quantity %> &times; <%= item.product.title %></li>
  <% end %>
</ul>
```



Handling Errors

- http://localhost:3000/carts/xpto



Handling Errors

- http://localhost:3000/carts/xpto

carts_contraoller.rb

```
13   # GET /carts/1
14   # GET /carts/1.json
15   def show
16     @cart = Cart.find(params[:id])
17
18     respond_to do |format|
19       format.html # show.html.erb
20       format.json { render :json => @cart }
21     end
22   end
```

If the cart cannot be found, **Active Record raises a RecordNotFound exception**, which we clearly need to handle.

We'll take two actions when an exception is raised:
- First, we'll **log the fact** to an internal log file using Rails' logger facility.2
- Second, we'll **redisplay the catalog page**, along **with a short message** to the user (something along the lines of "Invalid cart") so they can continue to use our site.

Handling Errors

- http://localhost:3000/carts/xpto

carts_contraoller.rb

```
def show
  begin
    @cart = Cart.find(params[:id])
  rescue ActiveRecord::RecordNotFound
    logger.error "Attempt to access invalid cart #{params[:id]}"
    redirect_to store_url, :notice => 'Invalid cart'
  else
    respond_to do |format|
      format.html # show.html.erb
      format.json { render :json => @cart }
    end
  end
end
```

We'll take two actions when an exception is raised:
- First, we'll **log the fact** to an internal log file using Rails' logger facility.2
- Second, we'll **redisplay the catalog page**, along **with a short message** to the user (something along the lines of "Invalid cart") so they can continue to use our site.

Handling Errors

- http://localhost:3000/carts/xpto

Finishing the Cart

- Implement the “empty cart”
- Calculate the total in the cart

Figure 10.5: Cart display with a total

Recommended readings

- **From the book "**Pragmatic Agile Web Development with Rails (4th Edition) by Sam Ruby, Dave Thomas and David Hanson, **up to page 256.**
- Check the main site: http://rubyonrails.org
  - http://guides.rubyonrails.org/active_record_validations_callbacks.html
  - http://guides.rubyonrails.org/testing.html
  - http://guides.rubyonrails.org/association_basics.html
  - http://guides.rubyonrails.org/active_record_querying.html