The RESTful Development and Rails Routing

CITS3403 Web & Internet Technologies

Reference: Ruby et al, Chapter 20

Include materials from Agile Web Development with Rails, 4th Ed. © 2012 The Pragmatic Bookshelf
Include materials from Ruby on Rails 3 Tutorial © 2011 Michael Hartl
Slides ©2011,2012 Wei Liu
Core of Rails Applications

- ActiveRecord
- Action Pack
  - ActionDispatch
  - ActionController
  - ActionView
REST: Representational State Transfer

- REST is an architectural style for developing distributed, networked systems and software applications such as the World Wide Web and web applications.

- The ideas behind REST were formalized in Chapter 5 of Roy Fielding’s 2000 PhD dissertation*.

Long-term state as Resources

• In a REST approach, servers communicate with clients using stateless connections. All the information about the state of the interaction between the two is encoded into the requests and responses between them. Long-term state is kept on the server as a set of identifiable resources.

• Clients access these resources using a well-defined (and severely constrained) set of resource identifiers (URLs in our context).

• REST distinguishes the content of resources from the presentation of that content.

• REST is designed to support highly scalable computing while constraining application architectures to be decoupled by nature.
What benefits does REST bring us?

• First, the formalities of a RESTful approach mean that network designers know when and where they can *cache* responses to requests. This enables load to be pushed out through the network, increasing performance and resilience while reducing latency.
Resources not services

- Second, the constraints imposed by REST can lead to easier-to-write (and maintain) applications.

- RESTful applications don’t worry about implementing remotely accessible services. Instead, they provide a regular (and simple) interface to a set of resources.
  - REST helps you make choices about which controllers and actions to write;
  - Your application implements a way of listing, creating, editing, and deleting each resource, and your clients do the rest.
REST in the context of Rails Applications

• Most application components such as (products, users, orders) are modelled as **resources** that can be **Created**, **Read**, **Updated** and **Deleted** - operations that corresponds to
  – the CRUD operations of relational databases, and
  – the four fundamental HTTP request methods: POST, GET, PUT and DELETE

• Rails constrains us to structure our applications a certain way using **MVC** (how to structure your source code)

• The **REST** philosophy tells us to structure the access to our applications too (how URLs map to code)

  **Constraints == Simplification**
1. The browser issues a request for the /users URI.
2. Rails routes /users to the index action in the Users controller.
3. The index action asks the User model to retrieve all the users (User.all)
4. The User model pulls all the users from the database.
5. The User model returns list of users to the controller.
6. The controller captures the users in the @users variable, which is passed to the index view.
7. The view uses Embedded Ruby to render the page as HTML.
8. The controller passes the HTML back to the browser.
A simple set of verbs operate on a rich set of nouns

- If we’re using HTTP, the **verbs** correspond to HTTP methods (GET, PUT, POST, and DELETE, typically).

- The **nouns** are the resources in our application. We name those resources using URLs.

- The Depot application contained a set of products. There are implicitly two resources here.
  - First, the individual products. Each constitutes a resource.
  - Second, the collection of products.
Concrete Example:

• To fetch a list of all the products, we could issue an HTTP GET request against this collection, say on the path /products.

• To fetch the contents of an individual resource, we have to identify it. The Rails way would be to give its primary key value (that is, its id). Again we’d issue a GET request, this time against the URL /products/1.

• To create a new product in our collection we use an HTTP POST request directed at the /products path, with the post data containing the product to add. Yes, that’s the same path we used to get a list of products. If you issue a GET to it, it responds with a list, and if you do a POST to it, it adds a new product to the collection.
The resources line in routes.rb

• Rails has direct support for this type of interface; it adds a kind of macro route facility, called \textit{resources}.

\begin{verbatim}
Depot::Application.routes.draw do |map|
  resources :products
end
\end{verbatim}

• The resources line caused seven new routes to be added to our application.

• It assumes the application will have a controller named \texttt{ProductsController}, containing seven actions with given names.
CRUD, HTTP Verbs and Actions

- The columns in the table below are (optional) route name, HTTP method, route path, and (on a separate line on this page) route requirements.

```ruby
>> rake routes

products GET /products(.:format) {:action=>"index", :controller=>"products"}
POST /products(.:format) {:action=>"create", :controller=>"products"}
new_product GET /products/new(.:format) {:action=>"new", :controller=>"products"}
edit_product GET /products/:id/edit(.:format) {:action=>"edit", :controller=>"products"}
product GET /products/:id(.:format) {:action=>"show", :controller=>"products"}
PUT /products/:id(.:format) {:action=>"update", :controller=>"products"}
DELETE /products/:id(.:format) {:action=>"destroy", :controller=>"products"}

Rails 4.x

products GET /products(.:format) products#index
POST /products(.:format) products#create
new_product GET /products/new(.:format) products#create
edit_product GET /products/:id/edit(.:format) products#edit
product GET /products/:id(.:format) products#show
PATCH /products/:id(.:format) products#update
PUT /products/:id(.:format) products#update
DELETE /products/:id(.:format) products#destroy
```
The Seven Actions, CRUD and HTTP Verbs

- **index**
  - Returns a list of the resources.

- **create**
  - Creates a new resource from the data in the POST request, adding it to the collection.

- **new**
  - Constructs a new resource and passes it to the client. This resource will not have been saved on the server. You can think of the new action as creating an empty form for the client to fill in.

- **show**
  - Returns the contents of the resource identified by params[:id].
The Seven Actions (cont.)

• **update**
  - Updates the contents of the resource identified by params[/:id] with the data associated with the request.

• **edit**
  - Returns the contents of the resource identified by params[/:id] in a form suitable for editing.

• **destroy**
  - Destroys the resource identified by params[/:id]
Paths and URLs

• Creating a resourceful route will also expose a number of helpers to the controllers in your application. In the case of `resources :products`:
  • `products_path` returns `/products`
  • `new_product_path` returns `/products/new`
  • `edit_product_path(:id)` returns `/products/:id/edit` (for instance, `edit_product_path(10)` returns `/products/10/edit`)
  • `product_path(:id)` returns `/products/:id` (for instance, `product_path(10)` returns `/products/10`)

• Each of these helpers has a corresponding `_url` helper (such as `products_url`) which returns the same path prefixed with the current host, port and path prefix.

• Because the router uses the HTTP verb and URL to match inbound requests, four URLs map to seven different actions.
Non-Resourceful Routing

• Static Routes
  – Useful for mapping legacy URLs
  – For example
    get
    'controller/:action/:id/with_user/:user_id'
  – would respond to /photos/show/1/with_user/2
  controller :sessions do
    get 'login' => :new
    post 'login' => :create
    delete 'logout' => :destroy
  end

http://guides.rubyonrails.org/routing.html#non-resourceful-routes
assert_generates(path, options, defaults={}, extras={}, message=nil)

- Verifies that the given set of options generates the specified path.

```ruby
def test_generates
  assert_generates("/", controller: "store",
                   action: "index")

  assert_generates("/products",
                   { controller: "products",
                     action: "index"})

  assert_generates("/line_items",
                   { controller: "line_items",
                     action: "create",
                     product_id: "1"},
                   { method: :post},
                   { product_id: "1"})
end
```
assert_recognizes(options, path, extras={}, message=nil)

- Verifies that routing returns a specific set of options given a path.

```ruby
def test_recognizes
  # Check the default index action gets generated
  assert_recognizes({"controller" => "store",
                     "action" => "index"}, "/")
  
  # Check routing to an action
  assert_recognizes({"controller" => "products",
                     "action" => "index"}, "/products")
  
  # And routing with a parameter
  assert_recognizes({"controller" => "line_items",
                     "action" => "create",
                     "product_id" => "1" },
                     
                     {path: "/line_items", method: :post},
                     
                     {"product_id" => "1"})
```

end
assert_routing(path, options, defaults={}, extras={}, message=nil)

- Combines the previous two assertions, verifying that the path generates the options and then that the options generate the path.

```ruby
def test_routing
  assert_routing("/", controller: "store",
                 action: "index")
  assert_routing("/products",
                 controller: "products",
                 action: "index")
  assert_routing({ path: "/line_items",
                  method: :post},
                  { controller: "line_items",
                    action: "create",
                    product_id: "1"},
                  {},
                  { product_id: "1"})
end
```