

appendix B:
Working with
Ruby on Rails

Since the beginning of the Web, both static and dynamic typed languages have been used for writing web applications. Java and its cousin C# have emerged as the eminent choices among the static typed languages. Meanwhile, the world of dynamic typed languages has seen a succession of popular choices: Perl, PHP, and now Ruby on Rails. To demonstrate Laszlo's ability to easily work with any web server, we presented a Java-based Struts web framework in appendix A. This appendix demonstrates how to interface Laszlo to Ruby on Rails (RoR) web framework.

B.1 What is Ruby on Rails?

RoR uses the Ruby language to implement Rails, which is an open source full-stack framework for building web applications. Full stack implies an integrated framework consisting of a web server, a model-view-controller (MVC)-based web framework for processing HTTP requests, and an object/relational mapping (ORM) tool for persisting data to a database. RoR promises significant increases in developer productivity by leveraging its runtime dynamics with *reflection* and *metaprogramming* to autogenerate many parts of a web application. This relieves developers from having to generate most of an application's plumbing code: controller, models, view, and unit tests. The end result is that developers can quickly get started working with a minimal application. They can then incrementally add and test new features.

B.1.1 The RoR approach

RoR's approach is based on a *convention-over-configuration* paradigm. RoR implies configuration based on the form of variable names used. Because these conventions are based on sound practice, it fits naturally within most people's working style. The practice of convention-over-configuration doesn't enforce any restrictions; it simply rewards consistent naming with increased autogeneration capabilities. Examples of these naming conventions are

- Model class names use "CamelCase" and are in singular form—e.g., `ShoppingCart`.
- Database names use underscores between words and are in plural form—e.g., `shopping_carts`.
- Primary keys uniquely identify rows in relational databases and are identified as `id`.
- Foreign keys join database tables and have an `id` suffix.

In this appendix, we start by creating an initial RoR development environment. The first task involves creating the product listings and adding pagination to their display. Next, we look at implementing session support that allows application state to be persisted across application lifecycles. So let's get started working with Ruby on Rails.

B.1.2 *Getting started with RoR*

Let's begin by creating a working Rails directory. In RoR, there are no naming conventions restricting application development to a particular directory; this allows us to choose any name we want. In a moment of heightened creativity, we chose "rails":

```
$ mkdir C:/rails
$ cd C:/rails
```

To build our application, we execute Rails with the application name as its argument:

```
$ rails store
```

This causes Rails to build its initial development environment, a self-contained environment including all the directories and files for our web application, as we can see with

```
$ cd store
$ ls
README    app          config  doc  log      script  tmp
Rakefile  components  db      lib  public  test    vendor
```

The command to start the WEBrick development server, located in the script directory, is executed by Ruby with the arguments shown here:

```
$ ruby script/server [-p]
```

By default the WEBrick server uses port 3000, but the `p` option allows other port numbers to be specified:

```
=> Booting WEBrick...
=> Rails application started on http://0.0.0.0:3000
=> Ctrl-C to shutdown server; call with --help for options
[2005-11-24 19:54:26] INFO  WEBrick 1.3.1
[2005-11-24 19:54:26] INFO  ruby 1.8.2 (2004-12-25) [i386-cygwin]
[2005-11-24 19:54:26] INFO  WEBrick::HTTPServer#start: pid=440 port=3000
```

A significant advantage of using the WEBrick server is that there is nothing to configure. Once the WEBrick startup message appears, we are ready to begin developing our RoR application.

B.1.3 Development with RoR

All RoR applications are stored in the app directory, which contains three directories (models, views, and controllers) conforming to the MVC (model-view-controller) states along with a helpers directory containing general-purpose utilities:

```
$ ls C:/rails/laszlomarket/app
  controllers/  helpers/  models/  views/
```

Rather than write an application, as we do with other systems, we execute Ruby commands to cause Rails to create significant portions of the application. Let's begin by adding a controller file to the Laszlo Market application to invoke its actions. We'll execute the ruby command with these arguments:

```
$ ruby script/generate controller laszlomarket
exists app/controllers/
exists app/helpers/
create app/views/laszlomarket
exists test/functional/
create app/controllers/laszlomarket_controller.rb
create test/functional/laszlomarket_controller_test.rb
create app/helpers/laszlomarket_helper.rb
```

In response to this command, RoR creates four files:

- An empty view directory, view/laszlomarket
- An empty controller file, controllers/laszlomarket_controller.rb, ready to be updated with the laszlomarket code
- The start of a functional test file, test/functional/laszlomarket_controller_test.rb
- An empty helpers file, helpers/store_helper.rb

The contents in these initial autogenerated files are relatively sparse, but RoR's ability to autogenerate development files will improve substantially in the next section, when we create a database schema for our product listing.

B.2 Providing a product listing

The product listing needs CRUD (create, replace, update, and delete) operations to provide easy administration of its contents. By adhering to RoR's convention-over-configuration approach, we'll allow RoR to autogenerate all this functionality. By the end of this section, we will have completed all of the RoR coding needed to support the product listing in the Laszlo Market. So let's get started with the database modifications.

B.2.1 Dealing with databases

Conforming to RoR's convention-over-configuration approach requires a few changes to our database setup. RoR expects to use three databases, called `store_dev`, `store_test`, and `store_prod`, for development, test, and production:

```
$ mysql -u root -p
Enter password: *****
Welcome to the MySQL monitor. Commands end with ; or \g.
mysql> create database store_dev;
mysql> create database store_test;
mysql> create database store_prod;
mysql> grant all on store_dev.* to laszlo@localhost;
mysql> grant all on store_test.* to laszlo@localhost;
mysql> grant all on store_prod.* to laszlo@localhost;
mysql> exit
```

Connecting with a database is one of the few places in RoR where a configuration file, `config/database.yml`, is required. Each of the three database environments must be configured with appropriate values to each of the fields displayed below to interface to the MySQL database. There are also testing and production sections, but we're showing only the development section:

```
development:
  adapter: mysql
  database: store_dev
  username: laszlo
  password: laszlo.2007
  host: 127.0.0.1
  port: 3306
```

After we complete this configuration, RoR is ready to interface to the MySQL database.

Plural database names

By convention, RoR's database names are specified in the plural, such as *products*, rather than *product*. So we need to tweak the Data Definition Language (DDL) statements from the Struts implementation to conform to this:

```
drop table if exists products;
create table products (
  id int not null auto_increment,
  sku  varchar(12) not null,
  title varchar(100) not null,
  description text not null,
  specs text not null,
  image varchar(200) not null,
  video varchar(200) not null,
```

```
category varchar(100) not null,  
price decimal(10,2) not null,  
primary key (id)  
);
```

Conforming to this convention allows the product model class to be mapped automatically to the database products table.

Building admin and store controllers

Web pages can generally be divided into two groups: those for inputting information and those for displaying information. Consequently, two controllers are required: an *admin controller* to handle the input side and a *store controller* to handle the display side. Let's start with the admin controller.

Scaffolding a content management system

Most applications require an interface for customers to enter information into a database. Although the bulk of a system's data is received through DDL statements executed on the database, customers still require an interface for data maintenance. This includes adding new records, deleting old records, and updating records. We can't expect our customers to write DDL statements, so we need a web-based CMS featuring list, show, new, and edit pages.

RoR can automatically build a simple but functional CMS through its metaprogramming services. Executing the generate command with a scaffold option causes RoR to generate the code for an admin controller for the products database table. This supports all the functionality for managing product content:

```
$ ruby script/generate scaffold Product Admin  
exists app/controllers/  
exists app/helpers/  
exists app/views/admin  
exists test/functional/  
dependency model  
exists app/models/  
exists test/unit/  
exists test/fixtures/  
identical app/models/product.rb  
identical test/unit/product_test.rb  
identical test/fixtures/products.yml  
create app/views/admin/_form.rhtml  
create app/views/admin/list.rhtml  
create app/views/admin/show.rhtml  
create app/views/admin/new.rhtml  
create app/views/admin/edit.rhtml  
create app/controllers/admin_controller.rb  
create test/functional/admin_controller_test.rb
```

```
create app/helpers/admin_helper.rb
create app/views/layouts/admin.rhtml
create public/stylesheets/scaffold.css
```

The newly created web pages that comprise the CMS system are indicated in bold. New database records are entered through the web page at the URL `http://localhost:3000/admin/new`. The other CMS commands, `edit`, `list`, and `show`, require a `product_id` parameter. This parameter can be specified like this: (`http://localhost:3000/admin/edit/1`) or (`http://localhost:3000/admin/edit?1`).

Figure B.1 shows the Rails-generated web pages, which allow customers to list, edit, or create products.

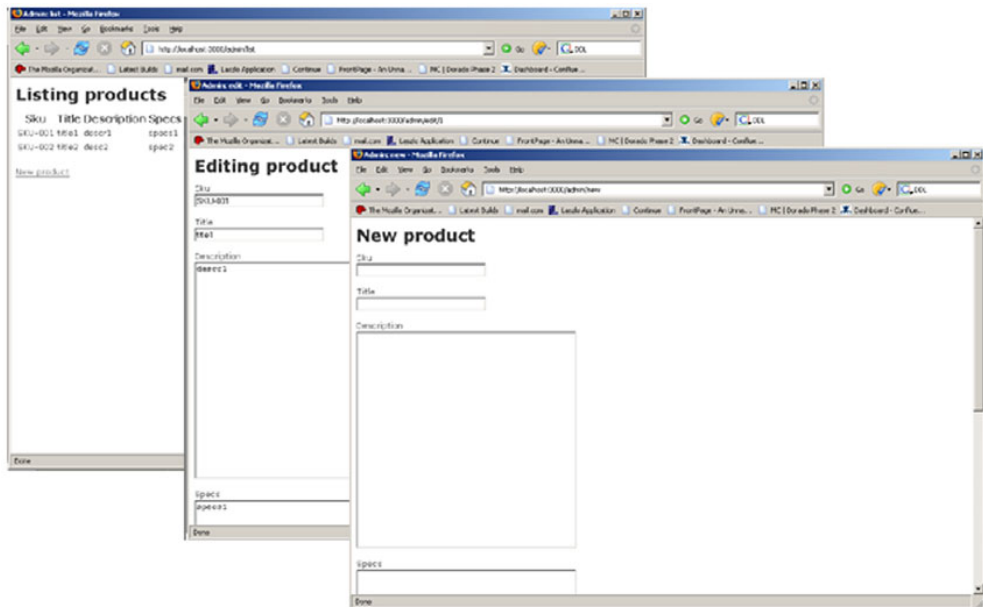


Figure B.1 This sequence of web pages allows a customer to easily add, modify, and delete product data records.

But before we can start entering data, we must add field validation to keep invalid values out of the database before they can cause errors.

Providing validation

The screens shown in figure B.1 require some validation specification before they are ready for use. This stops bad data from entering the system, which is easier than having to deal with it later. The following validations are required:

- Title, description, and image fields are mandatory.
- An image field must contain a full URL and terminate with a valid graphical suffix, png or jpg.
- The price must be greater than zero.

The `models/product.rb` file can be augmented with `validate` helper functions to perform the validation:

```
class Product < ActiveRecord::Base
  validates_presence_of
    :title, :description, :image ❶
  validates_numericality_of :price ❷

  private
    def validate
      errors.add(:price, "must be positive")
        unless price > 0.00 ❸
      end
    validates_format_of :image,
      :with => %r{^http:.\.(gif|jpg|png)$}i, :message =>
        "must be a URL for a JPG or PNG image" ❹
  end
```

First, we print an error message ❶ if a mandatory field—title, description, or image—is missing. Then, we make price ❷ a mandatory numerical field. The price field ❸ must contain a positive value. It is `private` to prevent outside classes from accessing it. Then, we validate the image field ❹, which must begin with `http:`, and terminate with a graphical suffix supported by both Laszlo Flash and DHTML.

This example demonstrates RoR's metaprogramming capabilities which, with the exception of some validation logic, are able to generate a basic CMS system for inputting data, with no manual coding required.

B.2.2 Building product XML output for Laszlo

While the admin controller is intended for displaying HTML, the store controller is intended for generating XML to communicate with Laszlo. To generate XML containing a complete list of the product items, start by having RoR create an initial implementation of the store controller:

```
$ ruby script/generate controller Store
```

Because the `scaffold` option isn't specified, only an empty `store_controller.rb` file is created. A `list` action is added to the store controller to create a listing of new products:


```

class StoreController < ApplicationController
  ...
  def list
    category = params[:category] ||= "new"
    @products = Product.find(:all,
                           :conditions=> ["category = ?", category])
  end
end

```

The `find` method results in a SQL call, where the `:conditions` parameter corresponds to a SQL where clause. A question mark placeholder allows dynamic SQL to be used. The result is an array of the matching rows whose default category is `new`. Other categories can be specified through a request parameter. Laszlo requires XML-over-HTTP, so the next step is to have RoR generate XML output.

Generating XML output

Next we need to create a *view file* that specifies a template to be expanded by RoR to generate the output. A template is a mixture of static text and code to generate dynamic content. RoR supports two template formats: RHTML and RXML.

With the admin controller, we can use RoR-generated RHTML files without modification. Each of these files is interpreted by an RHTML template to expand their inline expressions into dynamic data. The syntax for these inline expressions looks like JSP scriptlets with a delimiting set of `<%=` and `%>` markers.

The store controller uses RXML files to generate XML output. Rather than an RHTML template, a *builder* template interprets the contents of these files to expand a sequence of method calls into their corresponding XML elements. The method `xml.response` is converted into `<response>`. Node attributes can be specified using `=>` to identify the name and value. A method that converts to a parent node encloses a set of method calls—child nodes—within a pair of `do` and `end` keywords. The text node is indicated as a parameter to these methods. A collection created in the controller's action can be iterated with the `each` keyword. All the methods between the delimiting pair of `do` and `end` keywords are child nodes for each iteration.

Here's an example of XML output to populate the Laszlo Market's product list:

```

<response>
  <products>
    <product sku="SKU-001" title="Spiderman 2"
      price="19.99" image="dvd/In the cold.png">
      <description>When a CIA operation to purchase ...
    </description>
    <specs><p>Regional Code: 1 (US and Canada)<br>L ...
  </specs>

```

```

        <outline><b>Spiderman 2:</b><br/>When James Parker ...
      </outline>
    </product>
    ...
  </products>
</response>

```

To produce this XML output, the views/store/list.rxml file contains the following set of method calls:

```

xml.response() do
  xml.products() do
    @products.each do |product|
      xml.product(:sku => product.sku) do
        xml.id(product.id)
        xml.category(product.category)
        xml.price(product.price)
        xml.title(product.title)
        xml.image(product.image)
        xml.video(product.video)
        xml.description
          ("<![CDATA["+product.description+"]]")
        xml.outline("<![CDATA["+product.outline+"]]")
        xml.specs("<![CDATA["+product.specs+"]]")
      end
    end
  end
end

```

① Cycles through product records

In this code, an *iterator* ①, enclosed within the pair of vertical bars, works with a *code block*, nothing more than a chunk of code. The iterator operates on the values contained within the code block.

We'll next extend this XML output to support paged datasets.

B.2.3 Adding pagination

Pagination is easy, as it requires the addition of only a single line of code to the store controller:

```

def list
  category = params[:category] || "new"
  @products = Product.find(:all,
    :conditions=> ["category = ?", category])
  @product_pages, @products = paginate(:products,
    :per_page => 10, :order_by => 'title')
end

```

After all the products matching a particular category have been returned, a pagination helper, `paginate`, is applied. This expression uses the double assignment syntax (`x, y = paginate(products)`) to transform products into two objects:

paginator and `product_pages` (a collection of the products on the current page). `product_pages` is a paginator object that divides the products model objects into pages; each page contains ten rows sorted by their titles.

Additionally, we need to supply information that describes the contents of each returned page. This information consists of the current page number, the total number of pages, and a count of the total number of records. In the example page response, this information is contained within attributes of the parent node:

```
<response>
  <products page="0" totalpages="10" recordcount="98">
    <product id="1" sku="SKU-001" ... />
    ...
  </products>
</response>
```

The paginator and `product_pages` are used by the product view's `index.xml` file to create the paged output. This example shows how the paginator and `product_pages` are used to populate the paging-related attributes of the products node:

```
xml.response() do
  xml.products(:pages => @product_pages.current.offset,
               :totalpages => @product_pages,
               :recordcount => @product_count) do
    @products.each do |product|
      xml.product(:id => product.id,
                  :category => product.category,
                  :title => product.title,
                  :price => product.price,
                  :image => product.image) do
        xml.description(product.description)
        xml.outline(product.outline)
        xml.specs(product.specs)
      end
    end
  end
end
```

In the next section, we'll use sessions to maintain the shopping cart's state.

Handling sessions with Rails

To simplify session management, the store controller maintains a hash collection of key/value pairs called a *session*. Any information stored in the session is available to subsequent requests from the same browser. We'll use the session to store a `cart` object representing our shopping cart. We can retrieve this cart using the `get_cart` method, which is marked `private` to prevent it from being invoked through a URL. This method must occur at the end of the file, because a `private` access modifier causes all subsequent methods to be `private`:

```
private def get_cart
  session[:cart] ||= Cart.new
end
```

This method contains an unusual, but common, Ruby coding idiom that is equivalent to

```
if (session[:cart] != nil) session[:cart] = Cart.new
  return session[:cart]
```

or

```
session[:cart] = session[:cart] || Cart.new
```

The colon symbol serves as an identifier for a string of characters and specifies the cart key. The value of the session cart is checked for nil (RoR terminology for null). If it's nil, then a new cart instance is created. In Ruby, the last statement executed is returned from a method, allowing the return statement to be omitted. Although we have a cart object, it is empty. In the next section, we'll populate it with line item objects.

Creating a shopping cart

Our cart method specifies these methods for its parameter—identified with a colon:

```
class Cart
  attr_accessor :items

  def initialize
    @items = []
  end
end
```

First, attr_accessor specifies a set of getter/setter methods for items. Then, the initialize method is called by the constructor to establish the data types for the variables; items is an array.

When a new product is added to the shopping cart in the Laszlo Market, it generates a request to the server, resulting in the creation of an item. Items are maintained in memory and aren't persisted to the database until the entire order is completed and the purchase has been confirmed. When an item is persisted to a database, it is saved as a lineitem.

Creating items for the shopping cart

A lineitem object is a combination of a product and a quantity. When a customer purchases a video, the lineitem object consists of a product_id field to identify the product and a quantity field for the number of purchased products. The database schema for a line_items table looks like this:

```
create table line_items (
  id int not null auto_increment,
  product_id int not null,
  quantity int not null default 0,
  constraint fk_items_product foreign key (product_id)
    references products(id),
  primary key (id)
);
```

The `product_id` must be a foreign key in the `line_items` table, referring to the primary key `id` of the `Products` table. A foreign key allows database records to be linked together. MySQL requires the foreign key to be labeled as a constraint, since the `id` field from the `Products` table is constrained from being deleted. Attempting to delete this key would result in an integrity constraint error, since the foreign key would link to a nonexistent database record.

After creating this `line_items` database table, RoR is invoked to generate a `LineItem` model class, along with a couple of unit-testing classes. RoR performs reflection on the contents of this database table to create the `models/line_item.rb` file:

```
$ ruby script/generate model LineItem
exists app/models/
exists test/unit/
exists test/fixtures/
create app/models/line_item.rb
create test/unit/line_item_test.rb
create test/fixtures/line_items.yml
```

Since the `line_items` database table contains a foreign key, RoR adds a `belongs_to` method to this file to specify that the `lineitems` class have a parent/child relationship with the `products` class. The `lineitems` class is the child, because it contains a `C` column referencing the `id` column of the `Products` table:

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

Next we'll see how items are added to a shopping cart. These operations are initiated by the Laszlo Market application, which means they must be accessible as actions within RoR.

Adding an item to a cart

To make an action accessible through a URL, it is implemented as a method in the controller. As shown here, the `def` keyword identifies `add_to_cart` as a method. A method defaults to an access type of `public`, so it can be accessed through a URL:

```
class StoreController < ApplicationController
  def add_to_cart
    end
end
```

When a product is moved into the Laszlo shopping cart, it always has an initial quantity of one, which, of course, can be updated. This simplifies the requirements of the `add_to_cart` method, since the `quantity` parameter is always 1. It is called with a `product_id` parameter to add an item to the cart object; `add_to_cart` is accessed through the URL:

```
http://localhost:3000/store/add_to_cart/1
```

The method must find the product and its cart and build an `item` object to place into the cart. After it has updated these objects, processing is redirected to the `xcart` view to display the updated cart contents. If any errors are encountered, a rescue clause intercepts the exception to log the error and redirect processing to an error page:

```
class StoreController < ApplicationController
  def add_to_cart
    product = Product.find(params[:id])      ❶
    @cart = find_cart                         ❷
    @cart.add_product(product)               ❸
    @items = @cart.items
    render(:template => 'store/xcart')
  rescue                                     ❹
    logger.error("Failed to add product #{params[:id]}")
    flash[:error] =
      "Failed to add product: #{params[:id]}"  ❺
    render(:action => 'xerror')
  end
  ...
End
```

At line ❶, the corresponding item from the database, matching the `id` parameter, is found. Then the shopping cart for the session is obtained ❷ and updated ❸ with the new item. The `rescue` keyword ❹ in RoR specifies the types of exceptions that are handled. If no exception is specified and one is generated, an error message is written to the log and the error page is rendered. Rails maintains a special repository ❺ for error messages, called `flash`. The messages are available to the next request in the session before being automatically deleted.

The trickiest step is converting from a product object to an item object in the `for_project` method. This object is then added to the cart's items array:

```

class Cart
  def add_product(product)
    @items << LineItem.for_product(product)
  end
end

```

A `self` prefix names the object type of its caller:

```

class LineItem < ActiveRecord::Base
  belongs_to :product

  def self.for_product(product) ❶
    item = self.new ❷
    item.quantity = 1
    item.product = product
    item ❸
  end
end

```

Because the `for_product` method is called from a `LineItem` object, the `self` keyword ❶ refers to a `LineItem` object. When a constructor, such as `new`, is called ❷, it returns a `LineItem` object. After setting the properties for this object, its handle is returned ❸ to the caller.

Now we are ready to display the items in a shopping cart.

Producing cart XML output

In our code example, the render method is specified with a template and the path to a file `store/xcart.rxml`:

```

render(:template => 'store/xcart')

```

RoR comes with two template engines: RHTML, which works with HTML with embedded RoR statements, and Builder, which provides a more programmatic interface for constructing content. By convention, RoR uses Builder with all files having an `.rxml` extension. A builder template consists of a list of method names that are converted into XML statements.

The `items` array is made accessible to Builder, so each item can be iterated through to produce the complete item listing for the saved shopping cart:

```

xml.response() do
  xml.items() do
    @items.each do |item|
      xml.item(:id => item.product.id,
              :sku  => item.product.sku,
              :title => item.product.title,
              :image => item.product.image,
              :price => item.product.price) do

```

```

        xml.quantity(item.quantity)
      end
    end
  end
end

```

The following is sample output from the execution of this view:

```

<response>
  <items>
    <item sku="SKU-001" price="3.99"
      title="The Unfold"
      image=" dvd/unfold.png" quantity="1"/>
    ...
  </items>
</response>

```

Now that we can retrieve and display the contents of a sessioned shopping cart, let's look at the remaining operations necessary for updating and deleting items from it.

Updating an item in the shopping cart

After an item has been created and added to the cart, its quantity field can be updated through the Laszlo Market shopping cart interface. The URL that is generated by this action looks like this:

```
http://localhost:3000/store/update_product/1?quantity=10
```

The store controller handles this action through its `update_product` method, receiving a quantity parameter, whose value is available through `params[:quantity]`. After finding the product and cart, they are passed to the cart's `update_product` method. Finally, a status result is rendered in a view and sent back:

```

class StoreController < ApplicationController
  ...
  def update_product
    product = Product.find(params[:id])
    @cart = find_cart
    @cart.update_product(product, params[:quantity])
    render(:action => 'xstatus')
  rescue
    logger.error("Can't update product #{params[:id]}")
    flash[:error] = "Can't update product: #{params[:id]}"
    render(:action => 'xerror')
  end
end

```

The real work is done in the cart class. The `lineitem` object containing a matching `product_id` is found and its quantity field updated:


```

class Cart
  ...
  def update_product(product, quantity)
    quantity = quantity ? quantity : "1"
    item = @items.find {|i| i.product_id == product.id }
    raise "Can't find original product #{product.id}"
    if item.empty
      item.quantity = quantity.to_i
    end
  end
end

```

Defaults to 1

Returns multiple matches: illegal

The last step involves deleting an item from the sessioned shopping cart information.

Deleting a shopping cart item

An item is deleted from the cart object when a user drags one of the products from the Laszlo Market's shopping cart and drops it into the trash. The URL generated by this action looks like this:

```
http://localhost:3000/store/delete_product/1
```

The store controller handles this action through its `delete_product` method. After it finds the product and cart, they are passed to the cart's `delete_product` method. Finally, a status result is rendered in a view and sent back:

```

def delete_product
  product = Product.find(params[:id])
  @cart = find_cart
  @cart.delete_product(product)
  render(:template => 'store/status')
end

```

Because the items in the cart object reside only in memory, an item needs only to be removed from the items array. RoR supports a `find_all` method on arrays that works similarly to the Unix `grep`. To delete a specified item, the items array is set to all the items that don't match the `product_id` of a specified item. But RoR throws in one last little twist that makes this a bit tricky. Suppose that we write this code:

```

class Cart
  ...
  def delete_product(product)
    @items = @items.find_all {|i| i.product_id != product.id }
  end
end

```

RoR automatically invokes the getter method for a variable; it does not automatically invoke the setter method for the same variable. Rather, Ruby sees the assignment operator and decides that the name on the left must be a local variable, not a

method call to an attribute writer. To help Ruby make the correct determination, we replace instance variable `@items` with a `self` prefix:

```
self.items = @items.find_all { |i| i.product_id != product.id }
```

This back-end service is triggered in the Laszlo Market application when the user drags and drops a product from the shopping cart into the trashcan.

Handling errors

When an error occurs, a message is placed into the `flash[:error]` variable to make it easily accessible from a view error page. A temporary scratchpad for error messages, it is organized as a hash table so its values are accessed by name. Values stored into `flash` are available to the immediately following request. After that request has been processed, the values are removed. This is handy because `flash` variables need not be cleared when a new request arrives.

Flash information can be accessed in an error page through the `@flash` instance variable:

```
xml.response()  
  xml.status(@flash[:error], "error" => "true")  
end
```

This produces a simple XML-formatted error message that looks like this:

```
<results>  
  <status error="true">Invalid Product ID: 5</status>  
</results>
```

On the Laszlo side, the `node` attribute is checked for an error status, and if necessary, an error message is displayed.