

- Categories
- Database
- Postgres
- Interchange
- Ruby on Rails
- Spree
- Hosting
- Browsers
- Development
- Environments
- Version Control
- Perl
- SEO and Analytics
- CakePHP
- Django

# Spree Blog Archive

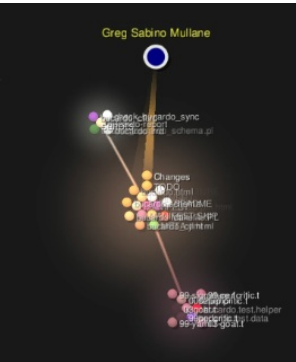
## Version Control Visualization and End Point in Open Source

Over the weekend, I **discovered** an open source tool for version control visualization, **Source**. I decided to put together a few videos to showcase End Point's involvement in several open source projects.

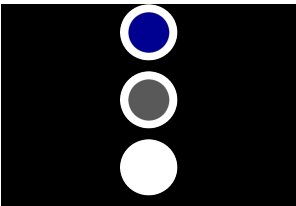
Here's a quick legend to help understand the videos below:



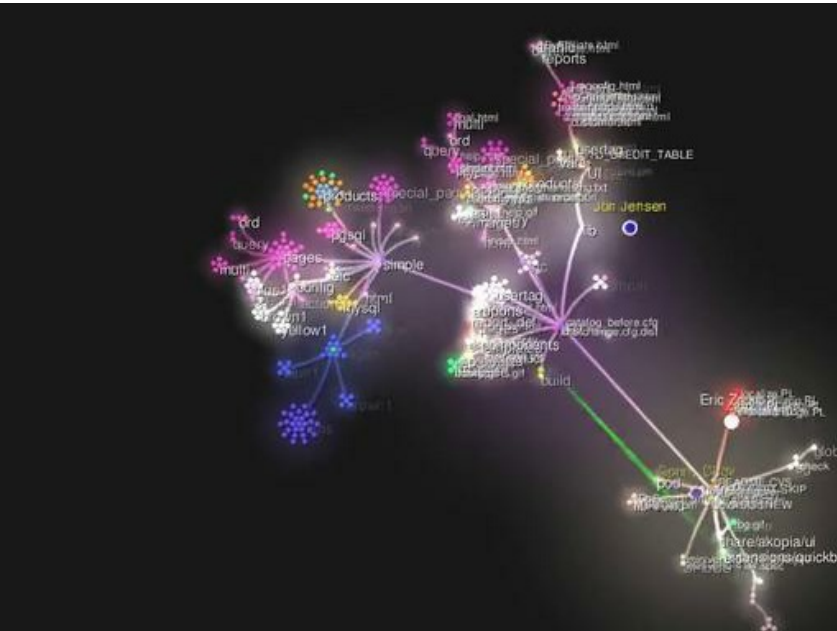
The branches and nodes correlate to directories and files, respectively. In the case of the image to the left, the repository has a main directory with several files and three directories. One of the child directories has one file and the other two have multiple files.



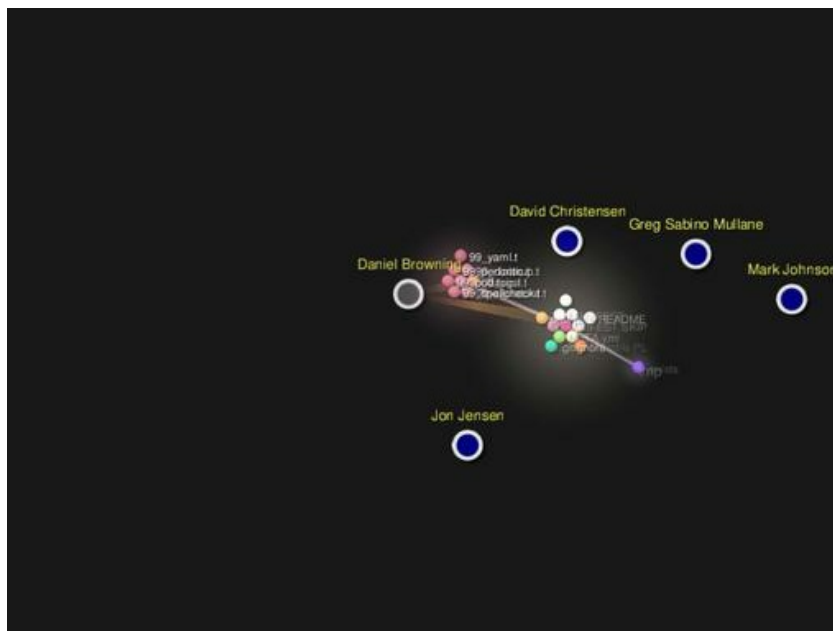
A big dot represents a person, and a flash connecting the person and a file signifies a commit.



- White + blue dots represent current End Point employees.
- White + grey dots represent former End Point employees.
- White dots represent other people, out there!



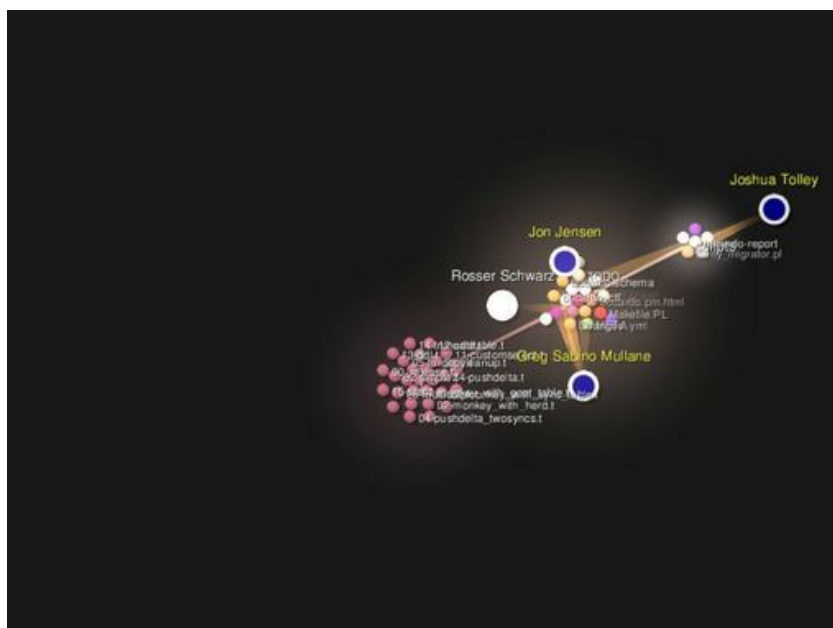
Interchange from endpoint on Vimeo.



**pgsi** from **endpoint** on **Vimeo**.



Spree from [endpoint](#) on [Vimeo](#).



**Bucardo** from **endpoint** on **Vimeo**.

One of the articles that references Gource suggests that the videos can be used to visualize and analyze the community involvement of a project (open source or not). One might also be able to qualitatively analyze the stability of project file architecture from a video, but this won't reveal anything definitive about the code stability since

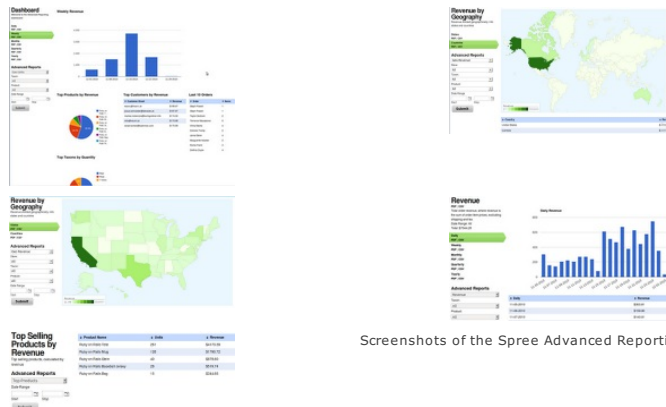
external factors can influence file structure. For example, since I am intimately familiar with the progress of Spree, I can identify when Spree transitioned to Rails 3 in the video, which required reorganization of the Spree core functionality (read more about this [here](#) and [here](#)).

In the case of this article, I wanted to highlight End Point's involvement in a few open source projects where we've had various levels of involvement. We've contributed to Interchange since 2000. We've been involved in Spree less lately, but had more presence in early 2009. In the smaller projects Bucardo and pgsi, End Point employees have worked on a team to be the primary contributors to the projects in addition to a few external contributors. Open source is important to End Point, and it's great to see our presence demonstrated in these cute videos.

## Dissecting a Rails 3 Spree Extension Upgrade

A while back, I wrote about the release of Spree 0.30.\* [here](#) and [here](#). I didn't describe extension development in depth because I hadn't developed any Rails 3 extensions of substance for End Point's clients. Last month, I worked on an advanced reporting extension for a client running on Spree 0.10.2. I spent some time upgrading this extension to be compatible with Rails 3 because I expect the client to move in the direction of Rails 3 and because I wanted the extension to be available to the community since Spree's reporting is fairly lightweight.

Just a quick rundown on what the extension does: It provides incremental reports such as revenue, units sold, profit (calculated by sales minus cost) in daily, weekly, monthly, quarterly, and yearly increments. It reports Geodata to show revenue, units sold, and profit by [US] states and countries. There are also two special reports that show top products and customers. The extension allows administrators to limit results by order date, "store" (for **Spree's multi-site architecture**), product, and taxon. Finally, the extension provides the ability to export data in PDF or CSV format using the Ruptor gem. One thing to note is that this extensions does not include new models - this is significant only because Rails 3 introduced significant changes to ActiveRecord, which are not described in this article.



To deconstruct the upgrade, I examined a git diff of the master and rails3 branch. I've divided the topics into Rails 3 and Spree specific categories.

### Rails 3 Specific

In my report extension, I utilize Ruptor's `_html` method, which returns a ruptor table object to an HTML table. With the upgrade to Rails 3, `ruptor.to_html` was spitting out escaped HTML with the addition of default XSS protection. The change is described [here](#), and required the addition of a new helper method (`raw`) to yield unescaped HTML:

```
diff --git a/app/views/admin/reports/top_base.html.erb b/app/views/admin/reports/top_base.html.erb
index 6cc6b70..92f2118 100644
--- a/app/views/admin/reports/top_base.html.erb
+++ b/app/views/admin/reports/top_base.html.erb
@@ -1,4 +1,4 @@
-<%= @report.ruptordata.to_html %>
+<%= raw @report.ruptordata.to_html %>
```

While troubleshooting the upgrade, I came across the following warning:

```
DEPRECATION WARNING: Using #request_uri is deprecated. Use fullpath instead. (called from ...)
```

I made several changes to address the deprecation warnings, did a full round of testing, and moved on.

```
diff --git a/app/views/admin/reports/_advanced_report_criteria.html.erb b/app/views/admin/reports/_advanced_report_criteria.html.erb
index ba69a2e..6d9c3f9 100644
--- a/app/views/admin/reports/_advanced_report_criteria.html.erb
+++ b/app/views/admin/reports/_advanced_report_criteria.html.erb
@@ -1,11 +1,11 @@
-<% @reports.each do |key, value| %>
-  <option <%= request.request_uri == "/admin/reports/#{key}" ? 'selected="selected"' : '' %>
-    value=<%= send("#{key}_admin_reports_url".to_sym) %>>
+  <option <%= request.fullpath == "/admin/reports/#{key}" ? 'selected="selected"' : '' %>
+    value=<%= send("admin_reports_#{key}_url".to_sym) %>>
-    <%= t(value[:name].downcase.gsub(" ", "_")) %>
+    <%= t(value[:name].downcase.gsub(" ", "_")) %>
```

An exciting change in Rails 3 is the advancement of Rails::Engines to allow easier inclusion of mini-applications inside the main application. In an ecommerce platform, it makes sense to break up the system components into Rails Engines. Extensions become gems in Spree and gems can be released through [rubygems.org](#). A gemspec is required in order for my extension to be treated as a gem by my main application, shown below. Componentizing elements of a larger platform into gems may become popular with the advancement of Rails::Engines / Railties.

```
diff --git a/advanced_reporting.gemspec b/advanced_reporting.gemspec
new file mode 100644
index 0000000..71f00a8
--- /dev/null
+++ b/advanced_reporting.gemspec
+Gem::Specification.new do |s|
+  s.platform = Gem::Platform::RUBY
+  s.name = 'advanced_reporting'
+  s.version = '2.0.0'
```

```

+ s.summary      = 'Advanced Reporting for Spree'
+ s.homepage     = 'http://www.endpoint.com'
+ s.author       = "Steph Skardal"
+ s.email        = "steph@endpoint.com"
+ s.required_ruby_version = '>= 1.8.7'
+
+ s.files        = Dir['CHANGELOG', 'README.md', 'LICENSE', 'lib/**/*', 'app/**/*']
+ s.require_path = 'lib'
+ s.requirements << 'none'
+
+ s.has_rdoc = true
+
+ s.add_dependency('spree_core', '>= 0.30.1')
+ s.add_dependency('ruport')
+ s.add_dependency('ruport-util') #, :lib => 'ruport/util'
+end

```

With the release of Rails 3, there was a major rewrite of the router and integration of rack-mount. The **Rails 3 release notes on Action Dispatch** provide a good starting point of resources. In the case of my extension, I rewrote the contents of config/routes.rb:

#### Before

```

map.namespace :admin do |admin|
  admin.resources :reports, :collection => {
    :sales_total => :get,
    :revenue     => :get,
    :units       => :get,
    :profit      => :get,
    :count       => :get,
    :top_products => :get,
    :top_customers => :get,
    :geo_revenue => :get,
    :geo_units  => :get,
    :geo_profit  => :get,
  }
  map.admin "/admin",
    :controller => 'admin/advanced_report_overview',
    :action => 'index'
end

```

#### After

```

Rails.application.routes.draw do
  #namespace :admin do
  #  resources :reports, :only => [:index, :show] do
  #    collection do
  #      get :sales_total
  #    end
  #  end
  #end
  match '/admin/reports/revenue' => 'admin/reports#revenue', :via => [:get, :post]
  match '/admin/reports/count' => 'admin/reports#count', :via => [:get, :post]
  match '/admin/reports/units' => 'admin/reports#units', :via => [:get, :post]
  match '/admin/reports/profit' => 'admin/reports#profit', :via => [:get, :post]
  match '/admin/reports/top_customers' => 'admin/reports#top_customers', :via => [:get, :post]
  match '/admin/reports/top_products' => 'admin/reports#top_products', :via => [:get, :post]
  match '/admin/reports/geo_revenue' => 'admin/reports#geo_revenue', :via => [:get, :post]
  match '/admin/reports/geo_units' => 'admin/reports#geo_units', :via => [:get, :post]
  match '/admin/reports/geo_profit' => 'admin/reports#geo_profit', :via => [:get, :post]
  match "/admin" => "admin/advanced_report_overview#index", :as => :admin
end

```

## Spree Specific

The biggest transition to Rails 3 based Spree requires extensions to transition to Rails Engines. In Spree 0.11.\*, the extension class inherits from Spree::Extension and the path of activation for extensions in Spree 0.11.\* starts in initializer.rb where the ExtensionLoader is called to load and activate all extensions. In Spree 0.30.\*, extensions inherit from Rails::Engine which is a subclass of Rails::Railtie. Making an extension a Rails::Engine allows it to hook into all parts of the Rails initialization process and interact with the application object. A Rails engine allows you run a mini application inside the main application, which is at the core of what a Spree extension is – a self-contained Rails application that is included in the main ecommerce application to introduce new features or override core behavior.

See the diffs between versions here:

#### Before

```

diff --git a/advanced_reporting_extension.rb b/advanced_reporting_extension.rb
deleted file mode 100644
index f75d967..0000000
--- a/advanced_reporting_extension.rb
+++ /dev/null
@@ -1,46 +0,0 @@
-# Uncomment this if you reference any of your controllers in activate
-# require_dependency 'application'
-
-
-
-
-class AdvancedReportingExtension < Spree::Extension
-  version "1.0"
-  description "Advanced Reporting"
-  url "http://www.endpoint.com/"
-
-  def self.require_gems(config)
-    config.gem "ruport"
-    config.gem "ruport-util", :lib => 'ruport/util'
-  end
-
-  def activate
-    Admin::ReportsController.send(:include, AdvancedReporting::ReportsController)
-    Admin::ReportsController::AVAILABLE_REPORTS.merge(AdvancedReporting::ReportsController::ADVANCED_REPORTS)
-
-    Ruport::Formatter::HTML.class_eval do
-      # Override some Ruport functionality
-    end
-  end
-end

```

#### After

```

diff --git a/lib/advanced_reporting.rb b/lib/advanced_reporting.rb
new file mode 100644
index 0000000..4e6fee6
--- /dev/null
+++ b/lib/advanced_reporting.rb
@@ -0,0 +1,50 @@

```

```

+require 'spree_core'
+require 'advanced_reporting_hooks'
+require "ruport"
+require "ruport/util"
+
+module AdvancedReporting
+  class Engine < Rails::Engine
+    config.autoload_paths += %W(#{config.root}/lib)
+
+    def self.activate
+      #Dir.glob(File.join(File.dirname(__FILE__), "../app/**/*.rb")) do |c|
+        # Rails.env.production? ? require(c) : load(c)
+      #end
+
+      Admin::ReportsController.send(:include, Admin::ReportsControllerDecorator)
+      Admin::ReportsController::AVAILABLE_REPORTS.merge(Admin::ReportsControllerDecorator::ADVANCED_REPORTS)
+
+      Ruport::Formatter::HTML.class_eval do
+        # Override some Ruport functionality
+      end
+    end
+
+    config.to_prepare &method(:activate).to_proc
+  end
+end

```

Rails Engines in Rails 3.1 will allow migrations and public assets to be accessed from engine subdirectories, but a work-around is required to access migrations and assets in the main application directory in the meantime. There are a few options for accessing Engine migrations and assets; Spree recommends a couple rake tasks to copy assets to the application root, shown here:

```

diff --git a/lib/tasks/install.rake b/lib/tasks/install.rake
new file mode 100644
index 0000000..c878a04
--- /dev/null
+++ b/lib/tasks/install.rake
@@ -0,0 +1,26 @@
+namespace :advanced_reporting do
+  desc "Copies all migrations and assets (NOTE: This will be obsolete with Rails 3.1)"
+  task :install do
+    Rake::Task['advanced_reporting:install:migrations'].invoke
+    Rake::Task['advanced_reporting:install:assets'].invoke
+  end
+
+  namespace :install do
+    desc "Copies all migrations (NOTE: This will be obsolete with Rails 3.1)"
+    task :migrations do
+      source = File.join(File.dirname(__FILE__), '..', '..', 'db')
+      destination = File.join(Rails.root, 'db')
+      puts "INFO: Mirroring assets from #{source} to #{destination}"
+      Spree::FileUtilz.mirror_files(source, destination)
+    end
+
+    desc "Copies all assets (NOTE: This will be obsolete with Rails 3.1)"
+    task :assets do
+      source = File.join(File.dirname(__FILE__), '..', '..', 'public')
+      destination = File.join(Rails.root, 'public')
+      puts "INFO: Mirroring assets from #{source} to #{destination}"
+      Spree::FileUtilz.mirror_files(source, destination)
+    end
+  end
+end

```

A minor change with the extension upgrade is a relocation of the hooks file. Spree hooks allow you to interact with core Spree views, described more in depth [here](#) and [here](#).

```

diff --git a/advanced_reporting_hooks.rb b/advanced_reporting_hooks.rb
deleted file mode 100644
index fcb5ab5..0000000
--- a/advanced_reporting_hooks.rb
+++ /dev/null
@@ -1,43 +0,0 @@
+class AdvancedReportingHooks < Spree::ThemeSupport::HookListener
+  # custom hooks go here
+end

```

```

diff --git a/lib/advanced_reporting_hooks.rb b/lib/advanced_reporting_hooks.rb
new file mode 100644
index 0000000..cca155e
--- /dev/null
+++ b/lib/advanced_reporting_hooks.rb
@@ -0,0 +1,3 @@
+class AdvancedReportingHooks < Spree::ThemeSupport::HookListener
+  # custom hooks go here
+end

```

A common behavior in Spree extensions is to override or extend core controllers and models. With the upgrade, Spree adopts the "decorator" naming convention:

```

Dir.glob(File.join(File.dirname(__FILE__), "../app/**/*.rb")) do |c|
  Rails.env.production? ? require(c) : load(c)
end

```

I prefer to extend the controllers and models with module includes, but the decorator convention also works nicely.

With the transition to Rails 3, I found that there were changes related to dependency upgrades. Spree 0.11.\* uses searchlogic 2.3.5, and Spree 0.30.1 uses searchlogic 3.0.0.\*. Searchlogic is the gem that performs the search for orders in my report to pull orders between a certain time frame or tied to a specific store. I didn't go digging around in the searchlogic upgrade changes, but I referenced Spree's core implementation of searchlogic to determine the required updates:

```

diff --git a/lib/advanced_report.rb b/lib/advanced_report.rb
@@ -13,11 +15,26 @@ class AdvancedReport
  self.params = params
  self.data = {}
  self.ruportdata = {}
+
+

```

```

+   params[:search] ||= {}
+   if params[:search][:created_at_greater_than].blank?
+     params[:search][:created_at_greater_than] =
+       Order.first(:order => :completed_at).completed_at.to_date.beginning_of_day
+   else
+     params[:search][:created_at_greater_than] =
+       Time.zone.parse(params[:search][:created_at_greater_than]).beginning_of_day rescue ""
+   end
+   if params[:search][:created_at_less_than].blank?
+     params[:search][:created_at_less_than] =
+       Order.last(:order => :completed_at).completed_at.to_date.end_of_day
+   else
+     params[:search][:created_at_less_than] =
+       Time.zone.parse(params[:search][:created_at_less_than]).end_of_day rescue ""
+   end
+
+   params[:search][:completed_at_not_null] ||= "1"
+   if params[:search].delete(:completed_at_not_null) == "1"
+     params[:search][:completed_at_not_null] = true
+   end
+   search = Order.searchlogic(params[:search])
+   search.checkout_complete = true
+   search.state_does_not_equal('canceled')
+
+   self.orders = search.find(:all)
+   self.orders = search.do_search
+
+   self.product_in_taxon = true
+   if params[:advanced_reporting]

```

Finally, there are substantial changes to the Rakefile, which are related to rake tasks and testing framework. These didn't impact my development directly. Perhaps when I get into more significant testing on another extension, I'll dig deeper into the code changes here.

```

diff --git a/Rakefile b/Rakefile
index f279cc8..f9e6a0e 100644
# lots of stuff

```

For those interested in learning more about the upgrade process, I recommend the reviewing the **Rails 3 Release Notes** in addition to reading up on Rails Engines as they are an important part of Spree's core and extension architecture. The advanced reporting extension described in this article is available [here](#).

## Speeding up the Spree demo site

There's a lot that can be done to speed up Spree, and Rails apps in general. Here I'm not going to deal with most of that. Instead I want to show how easy it is to speed up page delivery using standard HTTP server tuning techniques, demonstrated on [demo.spreecommerce.com](#).

First, let's get a baseline performance measure from the excellent [webpagetest.org](#) service using their remote Internet Explorer 7 tests:

- First page load time: **2.1 seconds**
- Repeat page load time: **1.5 seconds**

The repeat load is faster because the browser has images, JavaScript, and CSS cached, but it still has to check back with the server to make sure they haven't changed. Full details are [in this initial report](#).

The [demo.spreecommerce.com](#) site is run on a Xen VPS with 512 MB RAM, CentOS 5 i386, Apache 2.2, and Passenger 2.2. There were several things to tune in the Apache `httpd.conf` configuration:

- `mod_deflate` was already enabled. Good. That's a big help.
- Enable HTTP keepalive: `KeepAlive On` and `KeepAliveTimeout 3`
- Limit Apache children to keep RAM available for Rails: `StartServers 5`, `MinSpareServers 2`, `MaxSpareServers 5`
- Limit Passenger pool size to 2 child processes (down from the default 6), to queue extra requests instead of using slow swap memory: `PassengerMaxPoolSize 2`
- Enable browser & intermediate proxy caching of static files: `ExpiresActive On` and `ExpiresByType image/jpeg "access plus 2 hours"` etc. (see below for full example)
- Disable ETags which aren't necessary once Expires is enabled: `FileETag None` and `Header unset ETag`
- Disable unused Apache modules: free up memory by commenting out `LoadModule proxy`, `proxy_http`, `info`, `logio`, `usertrack`, `spelling`, `userdir`, `negotiation`, `vhost_alias`, `dav_fs`, `autoindex`, `most_authn_*` and `authz_*` modules
- Disable SSLv2 (for security and PCI compliance, not performance): `SSLProtocol all -SSLv2` and `SSLCipherSuite ALL:!ADH:!EXPORT56:RC4+RSA:+HIGH:+MEDIUM:-LOW:-SSLv2:-EXP`

After making these changes, without tuning Rails, Spree, or the database at all, a new [webpagetest.org](#) run reports:

- First page load time: **1.2 seconds**
- Repeat page load time: **0.4 seconds**

That's an easy improvement, a reduction of 0.9 seconds for the initial load and 1.1 seconds for a repeat load! Complete details are in [this follow-on report](#).

The biggest wins came from enabling HTTP keepalive, which allows serving multiple files from a single HTTP connection, and enabling static file caching which eliminates the majority of requests once the images, JavaScript, and CSS are cached in the browser.

Note that many of the resource-limiting changes I made above to Apache and Passenger would be too restrictive if more RAM or CPU were available, as is typical on a dedicated server with 2 GB RAM or more. But when running on a memory-constrained VPS, it's important to put such limits in place or you'll practically undo any other tuning efforts you make.

I wrote about these topics a year ago in a blog post about [Interchange ecommerce performance optimization](#). I've since expanded the list of MIME types I typically enable static asset caching for in Apache. Here's a sample configuration snippet to put in the `<VirtualHost>` container in `httpd.conf`:

```

ExpiresActive On
ExpiresByType image/gif "access plus 2 hours"
ExpiresByType image/jpeg "access plus 2 hours"
ExpiresByType image/png "access plus 2 hours"
ExpiresByType image/tiff "access plus 2 hours"
ExpiresByType text/css "access plus 2 hours"
ExpiresByType image/bmp "access plus 2 hours"
ExpiresByType video/x-flv "access plus 2 hours"
ExpiresByType video/mpeg "access plus 2 hours"
ExpiresByType video/quicktime "access plus 2 hours"
ExpiresByType video/x-ms-asf "access plus 2 hours"
ExpiresByType video/x-ms-wm "access plus 2 hours"
ExpiresByType video/x-ms-wmv "access plus 2 hours"
ExpiresByType video/x-ms-wmx "access plus 2 hours"
ExpiresByType video/x-ms-wvx "access plus 2 hours"

```

```
ExpiresByType video/x-msvideo "access plus 2 hours"
ExpiresByType application/postscript "access plus 2 hours"
ExpiresByType application/msword "access plus 2 hours"
ExpiresByType application/x-javascript "access plus 2 hours"
ExpiresByType application/x-shockwave-flash "access plus 2 hours"
ExpiresByType image/vnd.microsoft.icon "access plus 2 hours"
ExpiresByType application/vnd.ms-powerpoint "access plus 2 hours"
ExpiresByType text/x-component "access plus 2 hours"
```

Of course you'll still need to tune your Spree application and database, but why not tune the web server to get the best performance you can there?

## Spree on Rails 3: Part Two

Yesterday, I [discussed my experiences on getting Rails 3 based Spree](#) up and running. I've explained in several blog articles ([here](#) and [here](#)) that customizing Spree through extensions will produce the most maintainable code – it is not recommended to work directly with source code and make changes to core classes or views. Working through extension development was one of my primary goals after getting Spree up and running.

To create an extension named "foo", I ran `rails g spree:extension foo`. Similar to pre-Rails 3.0 Spree, a `foo` directory is created (albeit inside the `sandbox/` directory as a Rails Engine. The generator appends the `foo` directory details to the `sandbox/` Gemfile. Without the Gemfile update, the rails project won't include the new `foo` extension directory (and encompassed functionality). I reviewed the extension directory structure and files and found that `foo/lib/foo.rb` was similar to the the `*_extension.rb` file.

### New

```
require 'spree_core'

module Foo
  class Engine < Rails::Engine

    config.autoload_paths += %W(#{config.root}/lib)

    def self.activate
      # Activation logic goes here.
      # A good use for this is performing
      # class_eval on classes that are defined
      # outside of the extension
      # (so that monkey patches are not
      # lost on subsequent requests in
      # development mode.)
    end

    config.to_prepare &method(:activate).to_proc
  end
end
```

### Old

```
class FooExtension < Spree::Extension
  version "1.0"
  description "Describe your extension here"
  url "http://www.endpoint.com/"

  def activate
    # custom application functionality here
  end
end
```

I verified that the `activate` method was called in my extension with the following change:

```
require 'spree_core'

module Foo
  class Engine < Rails::Engine

    config.autoload_paths += %W(#{config.root}/lib)

    def self.activate
      Spree::BaseController.class_eval do
        logger.warn "inside base controller class eval"
      end
    end

    config.to_prepare &method(:activate).to_proc
  end
end
```

From here, [The Spree Documentation on Extensions](#) provides insight on further extension development. As I began to update an older extension, I ensured that `my_extension/lib/my_extension.rb` had all the necessary includes in the `activate` method and I copied over controller and library files to their new locations.

One issue that I came across was that migrations are not run with `rake db:migrate` and the public assets are not copied to the main project public directory on server restart. The documentation recommends building the migration within the application root (`sandbox/`), but this is not ideal to maintain modularity of extensions – each extension must include all of its migration files. To work-around this, it was recommended to copy over the install rake tasks from one of the core gems that copies migrations and public assets:

```
namespace :foo do
  desc "Copies all migrations and assets (NOTE: This will be obsolete with Rails 3.1)"
  task :install do
    Rake::Task['foo:install:migrations'].invoke
    Rake::Task['foo:install:assets'].invoke
  end
end

namespace :install do

  desc "Copies all migrations (NOTE: This will be obsolete with Rails 3.1)"
  task :migrations do
    source = File.join(File.dirname(__FILE__), '..', '..', 'db')
    destination = File.join(Rails.root, 'db')
    puts "INFO: Mirroring assets from #{source} to #{destination}"
    Spree::FileUtilz.mirror_files(source, destination)
  end

  desc "Copies all assets (NOTE: This will be obsolete with Rails 3.1)"
  task :assets do
    source = File.join(File.dirname(__FILE__), '..', '..', 'public')
    destination = File.join(Rails.root, 'public')
    puts "INFO: Mirroring assets from #{source} to #{destination}"
    Spree::FileUtilz.mirror_files(source, destination)
  end
end
```

After creating the extension based migration files and creating the above rake tasks, one would run the following from the application (`sandbox/`) directory:

```
steph@machine:/var/www/spree/sandbox$ rake foo:install
(in /var/www/spree/sandbox)
INFO: Mirroring assets from /var/www/spree/sandbox/foo/lib/tasks/../../db to /var/www/spree/sandbox/db
INFO: Mirroring assets from /var/www/spree/sandbox/foo/lib/tasks/../../public to /var/www/spree/sandbox/public

steph@machine:/var/www/spree/sandbox$ rake db:migrate
(in /var/www/spree/sandbox)
```

```
# migrations run
```

Some quick examples of differences in project setup and extension generation between Rails 3.\* and Rails 2.\*:

#### New

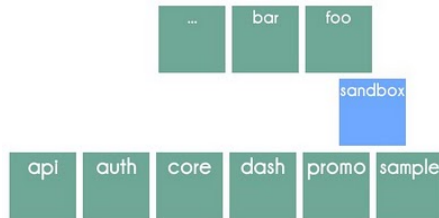
```
#clone project
#bundle install
rake sandbox
rails server
rails g spree:extension foo
rails g migration FooThing
```

#### Old

```
#clone project into "sandbox/"
rake db:bootstrap
script/server
script/generate extension Foo
script/generate extension_model Foo thing name:string start:date
```

Some of my takeaway comments after going through these exercises:

If there's anything I might want to learn about to work with edge Spree, it's Rails Engines. When you run Spree from source and use extensions, the architecture includes several layers of stacked Rails Engines:



Layers of Rails Engines in Spree with extensions.

After some quick googling, I found two helpful articles on Engines in Rails 3 [here](#) and [here](#). The Spree API has been inconsistent until now - hopefully the introduction of Rails Engine will force the API to become more consistent which may improve the extension community.

I didn't notice much deviation of controllers, models, or views from previous versions of Spree, except for massive reorganization. Theme support (including [Spree hooks](#)) is still present in the core. Authorization in Spree still uses authlogic, but I heard rumors of moving to devise eventually. The spree\_dash (admin dashboard) gem still is fairly lightweight and doesn't contain much functionality. Two fairly large code changes I noticed were:

- The checkout state machine has been merged into order and the checkout model will be eliminated in the future.
- The spree\_promo gem has a decent amount of new functionality.

Browsing through the [spree-user](#) Google Group might reveal that there are still several kinks that need to be worked out on edge Spree. After these issues are worked out and the documentation on edge Spree is more complete, I will be more confident in making a recommendation to develop on Rails 3 based Spree.

## Spree on Rails 3: Part One

A couple of weeks ago, I jumped into development on Spree on Rails 3. **Spree** is an open source Ruby on Rails ecommerce platform. End Point has been involved in Spree since its inception in 2008, and we continue to develop on Spree with a growing number of clients. Spree began to transition to Rails 3 several months ago. The most recent stable version of Spree (0.11.2) runs on Rails 2.\*, but the edge code runs on Rails 3. My personal involvement of Rails 3 based Spree began recently; I waited to look at edge Spree until Rails 3 had a bit of momentum and until Rails 3 based Spree had more documentation and stability. My motivation for looking at it now was to determine whether End Point can recommend Rails 3 based Spree to clients and to share insight to my coworkers and other members of the Spree community.

First, I looked at the messy list of gems that have built up on my local machine throughout development of various Rails and Spree projects. I found this simple little script to remove all my old gems:

```
#!/bin/bash

GEMS=`gem list --no-versions`
for x in $GEMS; do sudo gem uninstall $x --ignore-dependencies -a; done
```

Then, I ran `gem install rails` to install Rails 3 and dependencies. The following gems were installed:

```
abstract (1.0.0)
actionmailer (3.0.1)
actionpack (3.0.1)
activemodel (3.0.1)
activerecord (3.0.1)
activeresource (3.0.1)
activesupport (3.0.1)
arel (1.0.1)
builder (2.1.2)
bundler (1.0.2)
erubis (2.6.6)
i18n (0.4.1)
mail (2.2.7)
mime-types (1.16)
polyglot (0.3.1)
rack (1.2.1)
rack-mount (0.6.13)
rack-test (0.5.6)
rails (3.0.1)
railties (3.0.1)
rake (0.8.7)
thor (0.14.3)
treetop (1.4.8)
tzinfo (0.3.23)
```

Next, I cloned the Spree edge with the following command from [here](#):

```
git clone http://github.com/railsdog/spree.git
```

In most cases, developers will run Spree from the gem and not the source code ([see the documentation for more details](#)). In my case, I wanted to review the source code and identify changes. You might notice that the new spree core directory doesn't look much like the old one, which can be explained by the following: the Spree core code has been broken down into 6 separate core gems (api, auth, core, dash, promo, sample) that run as Rails Engines.

After checking out the source code, the first new task to run with edge Spree was `bundle install`. The bundler gem is installed by default in Rails 3. It works out of the box in Rails 3, and can work in Rails 2.3 with additional file and configuration changes. Bundler is a dependency management tool. Gemfile and Gemfile.lock in the Spree core specify which gems are required for the application. Several gems were installed with Spree's bundler configuration, including:



```

Installing webrat (0.7.2.beta.1)
Installing rspec-rails (2.0.0.beta.19)
Installing ruby-debug-base (0.10.3) with native extensions
Installing ruby-debug (0.10.3)
Installing state_machine (0.9.4)
Installing stringex (1.1.0)
Installing will_paginate (3.0.pre2)
Using spree_core (0.30.0.beta2) from source at /var/www/spree
Using spree_api (0.30.0.beta2) from source at /var/www/spree
Using spree_auth (0.30.0.beta2) from source at /var/www/spree
Using spree_dash (0.30.0.beta2) from source at /var/www/spree
Using spree_promo (0.30.0.beta2) from source at /var/www/spree
Using spree_sample (0.30.0.beta2) from source at /var/www/spree

```

The only snag I hit during **bundle install** was that the nokogiri gem required two dependencies be installed on my machine (libxslt-dev and libxml2-dev).

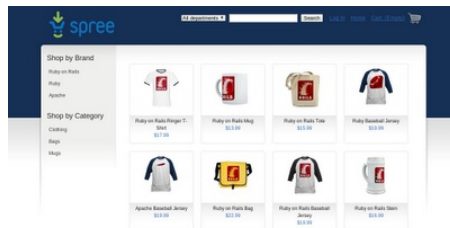
To create a project and run all the necessary setup, I ran **rake sandbox**, which completed the tasks listed below. The tasks created a new project, completed the basic gem setup, installed sample data and images, and ran the sample data bootstrap migration. In some cases, Spree sample data will not be used – the latter two steps can be skipped. The sandbox/ application directory contained a directory of folders that one might expect when developing in Rails (app, db, lib, etc.) and sandbox/ itself runs as a Rails Engine.

```

steph@machine:/var/www/spree$ rake sandbox
(in /var/www/spree)
  run rails new sandbox -GJT from "."
  append sandbox/Gemfile
  run rails g spree:site -f from "./sandbox"
  run rake spree:install from "./sandbox"
  run rake spree_sample:install from "./sandbox"
  run rake db:bootstrap AUTO_ACCEPT=true from "./sandbox"

```

After setup, I ran **rails server**, the new command for starting a server in Rails 3.\*, and verified my site was up and running.



Hooray - it's up!

There wasn't much to getting a Rails 3 application up and running locally. I removed all my old gems, installed Rails 3, grabbed the repository, allowed bundler to install dependencies and worked through one snag. Then, I ran my Spree specific rake task to setup the project and started the server. Tomorrow, I **share my experiences on extension development in Rails 3 based Spree**.

## Implementing Per Item Discounts in Spree

### Discounts in Spree

For a good overview of discounts in Spree, the documentation is a good place to start. The section on **Adjustments** is particularly apropos.

In general, the way to implement a discount in Spree is to subclass the Credit class and attach or allow for attaching of one or more Calculators to your new discount class. The Adjustment class file has some good information on how this is supposed to work, and the CouponCredit class can be used as a template of how to do such an implementation.

### What we Needed

For my purposes, I needed to apply discounts on a per Item basis and not to the entire order.

The issue with using adjustments as-is is that they are applied to the entire order and not to particular line items, so creating per line item discounts using this mechanism is not obviously straight forward. The good news is that there is nothing actually keeping us from using adjustments in this manner. We just need to modify a few assumptions.

### Implementation Details

This is going to be a high-level description of what I did with (hopefully) enough hints about what are probably the important parts to point someone who wants to do something similar in the same direction.

Analogous to the Coupon class in Spree, I create a Discount class. It holds the meta-data information about the discount. Specifically, the product that the discount applies to and the business logic for determining under what circumstances to apply the discount and how much to apply.

There is also a DiscountCredit class which subclasses the Credit class. In this class I re-define two methods:

- **applicable?** returns true when the discount applies to the line\_item
- **calculate\_adjustment** calculates the amount of the discount based on the business rules.

I also add a couple of convenience methods:

- **line\_item** returns self.adjustment\_source
- **discount** returns self.line\_item.variant.product.discount

The trick (as an astute reader might infer from the convenience methods) is to set the line\_item which the discount is getting applied to to the adjustment\_source attribute in the discount object.

The adjustment generally expects that you will be setting this to something like an instance of the Discount class, but as long as we ensure that LineItem implement any interface constraints required by Adjustments, we should be okay.

To that end, I monkey patch the LineItem class in my extension to add a method called add\_discount. This method creates a new instance of a DiscountCredit object and passes in itself as the adjustment\_source. I then add this credit object to the adjustments on the order.

I also add a method to iterate through all of the discounts to look for one that might already be applied to this line\_item instance. I use this method in the add\_discount method to ensure that I don't add more than one credit per line item.

To bring this together, I monkey patch the Order class to add a method that iterates through all of the line items in the order and calls `add_discount` on each one. I add a `after_save` callback which calls this method to ensure that discounts are applied to all line items each time the order is updated.

That takes care of the mechanics of applying the discounts. From this point several things will be taken care of by Spree. Any discounts that are not applicable will get removed. The cart totals will get added up properly and discounts will be applied as adjustments.

## Other things you might want to do

You may not want Spree to only display applied discounts at checkout as a (potentially) long list of credits tacked on to the end of the order.

For example, I found it useful to create some helpers to peek into the order adjustments and pull out the discount for a particular line item when displaying the cart. I also wanted to consolidate all of the discounts as a total amount under discount, rather than display them independently, so, I modified the views that handled displaying the credits.

In my implementation, I found it more straight-forward to forgo the use of calculators when implementing the business logic. But, they would work just fine as part of the Discount class and the `DiscountCredit#calculate_adjustment` method can call the `calculator#calculate` method to determine the amount to discount.

## Problems

This approach works because Spree automatically consolidates products/variants into the same `line_item` in the cart. In my approach, I assigned discounts at the product level, but applied them at the variant level. This worked for me because I didn't have any variants in my data set.

A general solution would probably assign discounts at the product level (it's too annoying to track them on a per-variant basis) and further track enough information to ensure that a discount was properly applied to any valid `line_items` that contained variants of that product.

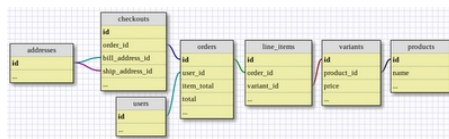
## Conclusion

All in all, I found that most of the heavy lifting was already done by the Adjustments code. All it really took was looking at the assumptions behind how the credits were working from a slightly different angle to see how I could modify things to allow per line item discounts to be implemented.

## Spree Sample Data: Orders and Checkout

A couple of months ago, I wrote about [setting up Spree sample data in your Spree project with fixtures](#) to encourage consistent feature development and efficient testing. I discussed how to create sample product data and provided examples of creating products, option types, variants, taxonomies, and adding product images. In this article, I'll review the sample order structure more and give an example of data required for a sample order.

The first step for understanding how to set up Spree order sample data might require you to revisit a simplified data model to examine the elements that relate to a single order. See below for the interaction between the tables orders, checkouts, addresses, users, line items, variants, and products. Note that the data model shown here applies to Spree version 0.11 and there are significant changes with Spree 0.30.



Basic diagram for Spree order data model.

The data model shown above represents the data required to build a single sample order. An order must have a corresponding checkout and user. The checkout must have a billing and shipping address. To be valid, an order must also have line items that have variants and products. Here's an example of a set of fixtures to create this bare minimum sample data:

```
#orders.yml
order_1:
  id: 1
  user_id: 1
  number: "R00000001"
  state: new
  item_total: 20.00
  created_at: Thu Jan 06 00:22:03 -0500 2011
  completed_at: Thu Jan 06 00:22:03 -0500 2011
  total: 20.00
  adjustment_total: 0.00

#addresses.yml
address_1:
  firstname: Steph
  lastname: Powell
  address1: 12360 West Carolina Drive
  city: Lakewood
  state_id: 889445952
  zipcode: 80228
  country_id: 214
  phone: 000-000-0000

#variants.yml
test_variant:
  product: test_product
  price: 10.00
  cost_price: 5.00
  count_on_hand: 10
  is_master: true
  sku: 1-master

#users.rb
#copy Spree core to create a user with id=1

#checkouts.yml
checkout_1:
  bill_address: address_1
  ship_address: address_1
  email: 'spree@example.com'
  order_id: 1
  ip_address: 127.0.0.1
  state: complete
  shipping_method: canada_post

#line_items.yml
li_1:
  order_id: 1
  variant: test_variant
  quantity: 2
  price: 10.00

#products.yml
test_product:
  name: Test Product 1
  description: Lorem ipsum...
  available_on: 2011-01-06 05:22:03
  count_on_hand: 10
  permalink: test-product
```

After adding fixtures for the minimal order data required, you might be interested in adding peripheral data to test custom work or test new feature development. This peripheral data might include:

- shipping methods: A checkout belongs to a shipping method, and has many shipping rates and shipments.
- shipments: An order has many shipments. Shipments are also tied to the shipping method.
- inventory units: An order has many inventory units, corresponding to each item in the order.
- payments: Orders and checkouts have many payments that must cover the cost of an order. Multiple payments can

- be assigned to each order.
- adjustments: Shipping charges and tax charges are tracked by adjustments, which belong to orders.
- return authorizations: Return authorizations belong to orders and track returns on orders, tied to inventory\_units in the order that are returned.

In my experience, I've worked with a few Spree projects where we created fixtures for setting peripheral sample data to test custom shipping and inventory management. Again, note that the data models described in this article are in place in Spree <= 0.11.0. Spree 0.30 will introduce data model changes to be discussed at a later date.

## Seeking a Ruby, Rails, Spree developer

Today I realized that we never posted our job announcement on our own blog even though we'd posted it to several job boards. So here it is:

We are looking for a developer who can consult with our clients and develop Ruby web applications. Most of our needs center around Rails, Spree, and SQL, but Sinatra, DataMapper, and NoSQL are lately coming into play too.

End Point is a 15-year-old web consulting company based in New York City, with 20 full-time staff developers working remotely from around the United States. We prefer open source technology and do collaborative development with Git, GNU Screen, IRC, and voice.

Experience with mobile and location-based technologies is a plus.

Please email [jobs@endpoint.com](mailto:jobs@endpoint.com) to apply.

## Learning Spree: 10 Intro Tips

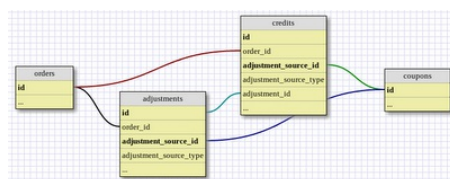
In climbing the learning curve with Spree development here are some observations I've made along the way:

1. **Hooks make view changes easier** — I was surprised at how fast I could implement certain kinds of changes because Spree's hook system allowed me to inject code without requiring overriding a template or making a more complicated change. Check out [Steph's](#) blog entries on hooks [here](#) and [here](#), and the [Spree documentation on hooks and themes](#).
2. **Core extensions aren't always updated** — One of the biggest surprises I found while working with Spree is that some Spree core extensions aren't maintained with each release. My application used the Beanstream payment gateway. Beanstream authorizations (without capture) and voids didn't work out of the box with Spree 0.11.0.
3. **Calculators can be hard to understand** — I wrote a custom shipping calculator and used calculators with coupons for the project and found that the data model for calculators was a bit difficult to understand initially. It took a bit of time for me to be comfortable using calculators in Spree. Check out the [Spree documentation on calculators](#) for more details.
4. **Plugins make the data model simpler after learning what they do** — I interacted with the plugins [resource\\_controller](#), [state\\_machine](#), and [will\\_paginate](#) in Spree. All three simplified the models and controllers interface in Spree and made it easier to identify the core behavior of Spree models and controllers.
5. **Cannot revert migrations** — Spree disables the ability to revert migrations due to complications with extensions which makes it difficult to undo simple database changes. This is more of a slight annoyance, but it complicated some aspects of development.
6. **Coupons are robust, but confusing** — Like calculators, the data model for coupons is a bit confusing to learn but it seems as though it's complicated to allow for robust implementations of many kinds of coupons. [Spree's documentation on coupons and discounts](#) provides more information on this topic.
7. **Solr extension works well** — I replaced Spree's core search algorithm in the application to allow for customization of the indexed fields and to improve search performance. I found that the [Solr extension for Spree](#) worked out of the box very well. It was also easy to customize the extension to perform indexation on additional fields. The only problem is that the Solr server consumes a large amount of system resources.
8. **Products & Variants** — Another thing that was a bit strange about Spree is that every product has at least one variant referred to as the master variant that is used for baseline pricing information. Spree's data model was foreign to me as most ecommerce systems I've worked with have had a much different product and variant data model.
9. **Routing** — One big hurdle I experienced while working with Spree was how Rails routing worked. This probably stemmed from my inexperience with the resource\_controller plugin, or from the fact that one of the first times I worked with Rails routing was to create routes for a nested resource. Now that I have learned how routing works and how to use it effectively, I believe it was well worth the initial struggle.
10. **Documentation & Community** — I found that the [documentation for Spree](#) was somewhat helpful at times, but the [spree-user Google group](#) was more helpful. For instance, I got a response on Beanstream payment gateway troubleshooting from the Spree extension author fairly quickly after asking on the mailing list.

I believe that Spree is an interesting project with a somewhat unusual approach to providing a shopping cart solution. Spree's approach of trying to implement 90% of a shopping cart system is very different from some other shopping cart systems which overload the code base to support many features. The 90% approach made some things easier and some things harder to do. Things like hooks and extensions makes it far easier to customize than I expected it would be, and it also seems like it helps avoid the build up of spaghetti code which comes from implementing a lot of features. However, allowing for a "90%" solution seems to make some things like calculators a bit harder to understand when getting started with Spree, since the implementation is general and robust to allow for customization.

## Spree: Gift Certificates and Coupons

In a recent Spree project, I've been working with [Bill](#) to add gift certificate functionality. According to the [Spree documentation](#), gift certificate functionality is trivial to implement using the existing coupon architecture. Here are some of the changes we went through as we tried to use the coupon architecture for gift certificate implementation - we found that it wasn't so simple after all.



Here is a very simplified visualization of the coupon and adjustment data model in Spree. Coupons use polymorphic calculators to compute the applicable discount.

First, Bill and I brainstormed to come up with an initial set of changes required for implementing gift certificates as coupons after we reviewed the data model shown above:

1. Add logic to create a coupon during checkout finalization, which was done with the following:

```
# coupon object class method
+def self.generate_coupon code
```

```

+ # some method to generate an unused random coupon code beginning in 'giftcert-'
+end

# inside order model during checkout finalization
+line_items.select { |li| li.variant.product.is_gift_cert? }.each do |line_item|
+ line_item.quantity.times do
+   coupon = Coupon.create(:code => Coupon.generate_coupon_code,
+                           :description => "Gift Certificate",
+                           :usage_limit => 1,
+                           :combine => false,
+                           :calculator => Calculator::FlatRate.new)
+   coupon.calculator.update_attribute(:preferred_amount, line_item.variant.price)
+ end
+end

```

2. Add logic to decrease a coupon amount during checkout finalization if used:

```

# order model during checkout finalization
+coupon_credits.select{ |cc| cc.adjustment_source.code.include?('giftcert-') }.each do |coupon_credit|
+ coupon = coupon_credit.adjustment_source
+ amount = coupon.calculator.preferred_amount - item_total
+ coupon.calculator.update_attribute(:preferred_amount, amount < 0 ? 0 : amount)
+end

```

3. Add relationship between line item and coupon because we'd want to have a way to associate coupons with line items. The intention here was to limit a gift certificate line item to a quantity of 1 since the gift certificate line item might include personal information like an email in the future.

```

+LineItem.class_eval do
+ has_one :line_item_coupon
+ has_one :coupon, :through => :line_item_coupon
+end

+class LineItemCoupon < ActiveRecord::Base
+ belongs_to :line_item
+ belongs_to :coupon
+
+ validates_presence_of :line_item_id
+ validates_presence_of :coupon_id
+end

```

4. Create the sample data for a gift certificate (coupon) - the implementation offers a master variant for a fixed cost of \$25.00. In addition to the code below, Bill created sample data to assign a product property is\_gift\_cert to the product.

```

# products.yml
+gift_certificate:
+ name: Gift Certificate
+ description: Gift Certificate
+ available_on: 2011-01-06 05:22:03
+ permalink: gift-certificate
+ count_on_hand: 100000

# variants.yml
+gift_cert_variant:
+ product: gift_certificate
+ sku: giftcert
+ price: 25.00
+ is_master: true
+ count_on_hand: 10000
+ cost_price: 25.00
+ is_extension: false

```

5. Finally, Bill edited the order mailer view to include gift certificate information

After the above changes were implemented, additional changes were required for our particular Spree application.

1. Adjust the shipping API so it doesn't include gift certificates in the shipping request, because gift certificates aren't shippable. Below is an excerpt of the XML builder code that generates the XML request made to the shipping API:

```

# shipping calculator
~order.line_items.each do |li|
+order.line_items.select { |li| !li.variant.product.is_gift_cert? }
  x.item {
    x.quantity(li.quantity)
    x.weight(li.variant.weight != 0.0 ? li.variant.weight : Spree::MyShipping::Config[:default_weight])
    x.length(li.variant.depth ? li.variant.depth : Spree::MyShipping::Config[:default_depth])
    x.width(li.variant.width ? li.variant.width : Spree::MyShipping::Config[:default_width])
    x.height(li.variant.height ? li.variant.height : Spree::MyShipping::Config[:default_height])
    x.description(li.variant.product.name)
  }
}

```

2. Create a new calculator for free shipping applicable to orders with gift certificate line items only, using the is\_gift\_cert product property:

```

# registering the calculator inside Spree site_extension.rb (required for all calculators to be used in Spree)
+[
+ Calculator::GiftCertificateShipping,
+]
+.each { |c_model|
+ begin
+   c_model.register if c_model.table_exists?
+ rescue Exception => e
+   $stderr.puts "Error registering calculator #{c_model}"
+ end
+}

# shipping method and calculator creation in sample data
+s = ShippingMethod.new(:zone_id => 16, :name => 'Gift Certificate Shipping')
+s.save
+c = Calculator.new
+c.calculable = s
+c.type = 'Calculator::GiftCertificateShipping'
+c.save

# calculator for free gift cert shipping
+class Calculator::GiftCertificateShipping < Calculator
+ ...
+ def available?(order)

```

```

+   order.line_items.inject(0) { |sum, li| sum += li.quantity if !li.variant.product.is_gift_cert?; sum } == 0
+ end
+
+ def compute(line_items)
+   0
+ end
+end

```

After Bill implemented these changes, I contemplated the following code more:

```

coupon_credits.select{ |coupon_credit| coupon_credit.adjustment_source.code =~ /^giftcert-/}.each do |coupon_credit|
  coupon = coupon_credit.adjustment_source
  amount = coupon.calculator.preferred_amount - item_total
  coupon.calculator.update_attribute(:preferred_amount, amount < 0 ? 0 : amount)
end

```

I wondered why the coupon amount being decremented by the item\_total and not the order total. What about shipping and sales tax? I verified by looking at the the Spree Coupon class that a coupon's amount will only take into account the item total and not shipping or tax, which would present a problem since gift certificates traditionally apply to tax and shipping costs.

Order Summary	
Item Total:	\$15.99
Tax:	\$0.00
Shipping (UPS Two Day):	\$10.00
Coupon (SPREE):	(\$-15.99)
<b>Order Total:</b>	<b>\$0.00</b>

Coupon Code

In the Spree core, coupons are never applied to shipping or tax costs.

I investigated the following change to separate coupon and gift certificate credit calculation:

```

+def site_calculate_coupon_credit
+  return 0 if order.line_items.empty?
+  amount = adjustment_source.calculator.compute(order.line_items).abs
+  order_total = adjustment_source.code.include?('giftcert-') ? order.item_total + order.charges.total : order.item_total
+  amount = order_total if amount > order_total
+  -1 * amount
+end

```

After this change, I found that when arriving on the payment page where the gift certificate has covered the entire order including tax and shipping, the payment logic isn't set up handle orders with a total cost of 0. Additional customization on payment implementation, validation and checkout flow would be required to handle orders where gift certificates cover the entire cost. However, rather than implementing these additional customizations, our client was satisfied with the implementation where gift certs don't cover tax and shipping, so I did not pursue this further.

In the future, I'd recommend creating a new model for gift certificate and gift certificate credit management rather than combining the business logic with coupons, because:

1. The coupon implementation in Spree doesn't have a whole lot to it. It uses several custom Spree calculators, has a backend CRUD interface, and credits are applied to orders. Grabbing the coupon implementation and copying and modifying it for gift certificates shouldn't be daunting.
2. It will likely be more elegant to separate coupon logic from gift certificate logic. Coupons and gift certificates share a few business rules, but not all. Gift certificates traditionally apply to tax and shipping and multiple gift certificates can be used on one order (but this part can be configurable). Coupons may have more complex logic to apply to items and do not traditionally get applied to tax and shipping (however, in some cases a free shipping coupon may be needed that covers the cost of shipping only). Additionally, a big difference in business logic is that gift certificates should probably be treated as a payment, where checkout accepts gift certificates as a form of payment, and the backend provides reporting on the gift certificate driven payments. Rather than dirtying-up the the coupon logic with checks for gift certificates versus coupon behavior, it'll be more elegant to separate the logic into classes that address the individual business needs.

Besides "hindsight is 20/20", the takeaway for me here is that you have to understand business rules and requirements for coupon and gift certificate implementation in ecommerce, which can get tricky quickly. We were lucky because the client was satisfied with the resulting behavior of using the coupon architecture for gift certificates. Hopefully, the takeaway for someone not familiar with Spree is that gift certificate implementation might require things like functionality for creating gift certificates after checkout completion, decrementing the gift certificate after it's used, backend reporting to show gift certificates purchase and use and coding for the impact of gift certificate purchase on shipping.

Note that all of the changes described here apply to the latest stable version of Spree (0.11.0). After taking a look at the Spree edge code, I'll mention that there is a bit of an overhaul on coupons (to be called promotions). However, it looks many of the customizations described here would be needed for gift certificate implementation as the edge promotions still apply to item totals only and do not include any core modifications in accepting a credit as a payment.

Learn more about End Point's [Ecommerce Development](#) or [Ruby on Rails Ecommerce Services](#).

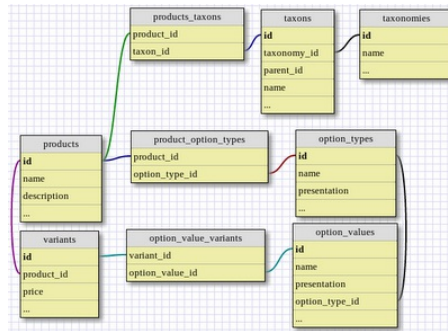
## Spree: Working with Sample Product Data

It's taken me a bit of time to gain a better understanding of working with Spree sample data or **fixtures**, but now that I am comfortable with it I thought I'd share some details. The first thing you might wonder is why should you even care about sample data? Well, in our project, we had a few motivations for creating sample data:

1. **Multiple developers, consistent sample data provides consistency during development.** End Point offers **SpreeCamps**, a hosting solution that combines the open source Spree technology with devcamps to allow multiple development and staging instances of a Spree application. In a recent project, we had a two developers working on different aspects of the custom application in SpreeCamps; creating meaningful sample data allowed each developer to work from the same data starting point.
2. **Unit testing.** Another important element of our project includes adding unit tests to test our custom functionality. Consistent test sample data gave us the ability to test individual methods and functionality with confidence.
3. **Application testing.** In addition to unit testing, adding sample data gives the ability to efficiently test the application repeatedly with fresh sample data.

Throughout development, our standard practice is to repeatedly run `rake db:bootstrap SKIP_CORE=1 AUTO_ACCEPT=1`. Running bootstrap with these arguments will not create Spree's core sample data set, but it will set the core's default

data that includes some base zones, zone members, countries, states, and roles, data that is essential for the application to work.



Product data relationship in Spree

The first data I create is the product data. As you can see from the image above, products data may have relationships with other tables including option\_types, option\_values, and taxons, or the tables that create has and belongs to many relationships between products and these elements.

The most simple form of sample data might include one test product and its master variant, shown below. If you do not define a master variant for a product, the product page will crash, as a master variant is required to display the product price.

```
products.yml
test_product:
  id: 1
  name: Test Product
  description: Lorem ipsum...
  available_on: 2011-01-06 05:22:03
  count_on_hand: 10
  permalink: test-product
```

```
variants.yml
test_variant:
  product: test_product
  price: 10.00
  cost_price: 5.00
  count_on_hand: 10
  is_master: true
  sku: 1-master
```

To expand on this, you might be interested in adding option types and values to allow for sizes to be assigned to this variant, shown below. The option type and option value data structure provides a flexible architecture for creating product variants, or multiple *varieties* of a single product such as different sizes, colors, or combinations of these option values.

```
option_types.yml
size:
  name: size
  presentation: Size
```

```
option_values.yml
small:
  name: Small
  presentation: Small
  option_type: size
large:
  name: Large
  presentation: Large
  option_type: size
```

```
product_option_types.yml
test_product_size:
  product: test_product
  option_type: size
```

```
variants.yml
# ... master variant
small variant:
  product: test_product
  option_values: small
  price: 10.00
  cost_price: 5.00
  count_on_hand: 10
  sku: 1-small
large variant:
  product: test_product
  option_values: large
  price: 20.00
  cost_price: 10.00
  count_on_hand: 10
  sku: 1-large
```

Another opportunity for expansion on sample product data is the taxonomy structure, which is very flexible. A root taxonomy can be thought of as a tree trunk with branches; products can be assigned to any number of branches. If we assume you have multiple test products, you might set up the following test data:

```
taxonomies.yml
category:
  name: Category
brand:
  name: Brand
```

```
taxons.yml
category_root:
  id: 1
  name: Category
  taxonomy: category
  permalink: c/
jackets:
  id: 2
  name: Jackets
  taxonomy: category_root
  permalink: c/jackets/
  parent_id: 1
  products: test_product, test_product2, test_product3
pants:
  id: 3
  name: Pants
  taxonomy: category_root
  permalink: c/pants/
  parent_id: 1
  products: test_product4, test_product5, test_product6
brand_root:
  id: 4
  name: Brand
  taxonomy: brand
  permalink: b/
brand_one:
  id: 5
```

```

name: Brand One
taxonomy: brand
permalink: b/brand-one/
parent_id: 4
products: test_product, test_product3, test_product5
brand_two:
id: 6
name: Brand Two
taxonomy: brand
permalink: b/brand-two/
parent_id: 4
products: test_product2, test_product4, test_product6

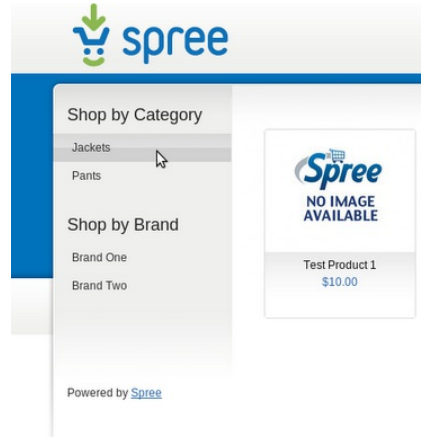
```

I also needed to include the `taxons.rb` (used in the Spree core sample data) to assign products to taxons correctly.

```

taxons.rb
Taxon.rebuild!
Taxon.all.each{|t| t.send(:set_permalink); t.save}

```



Example taxonomies created with Spree sample data

My last step in creating Spree sample data is to add product images. I've typically added the image via the Spree backend first, and then copied the images my site extension directory.

```

# upload Spree images

# create sample image directory
mkdir RAILS_ROOT/vendor/extensions/site/lib/tasks/sample/

# copy the uploaded images to the sample image directory
cp -r RAILS_ROOT/public/assets/products/ RAILS_ROOT/vendor/extensions/site/lib/tasks/sample/

```

After uploading and copying the images over, I include the image information in `assets.yml`. The ID for each asset must be equal to the directory containing the multiple image sizes. For example, the directory `RAILS_ROOT/vendor/extensions/site/lib/tasks/sample/1/` contains directories `original`, `large`, `product`, `small`, and `mini` with images sized respectively.

```

assets.yml
il:
  id: 1
  viewable: test_product
  viewable_type: Product
  attachment_content_type: image/jpg
  attachment_file_name: blue_sky.jpg
  attachment_width: 1024
  attachment_height: 683
  type: Image
  position: 1

```

And finally, I use a modified version of the Spree's core `products.rb` file to copy over product images during bootstrap:

```

products.rb
require 'find'
# make sure the product images directory exists
FileUtils.mkdir_p "#{RAILS_ROOT}/public/assets/products/"

# make product images available to the app
target = "#{RAILS_ROOT}/public/assets/products/"
source = "#{RAILS_ROOT}/vendor/extensions/site/lib/tasks/sample/products/"

Find.find(source) do |f|
  # omit hidden directories (SVN, etc.)
  if File.basenames(f) =~ /^[\.]/
    Find.prune
    next
  end

  src_path = source + f.sub(source, '')
  target_path = target + f.sub(source, '')

  if File.directory?(f)
    FileUtils.mkdir_p target_path
  else
    FileUtils.cp src_path, target_path
  end
end

```

With my sample data defined in my Spree site extension, I run `rake db:bootstrap SKIP_CORE=1 AUTO_ACCEPT=1` to create the above products, variants, and taxonomy structure. I commit my changes to the git repository, and other developers can work with the same set of products, variants, taxonomies including product images. During development, I also add unit tests to test model methods that interact with our sample data. An alternative to setting up Spree sample data described in this article is to dump entire databases and reimport them and manage the sample images manually, but I find that the approach described here forces you to understand the Spree data model better.





Sample product image created with the sample data above.

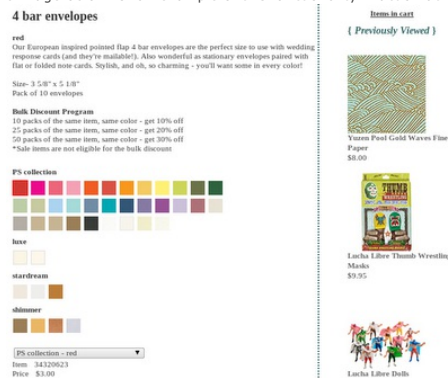
In addition to setting up sample product data, I've worked through creating sample orders, shipping configuration, and tax configuration. I hope to discuss these adventures in the future.

Learn more about End Point's [Ecommerce Development](#) or [Ruby on Rails Ecommerce Services](#).

## Spree vs Magento: Feature List Revisited

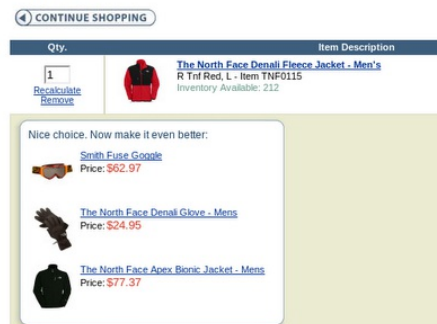
A little over a month ago, I wrote an article on [Spree vs Magento Features](#). Recently, a client asked me to describe the features mentioned in that article. I thought this was another great opportunity to expand on my response to the client. So, here I am, revisiting ecommerce features in Spree and Magento. The [original article](#) can be referenced to compare availability of these features in Spree and Magento.

- **Product reviews and/or ratings:** functionality to allow customers to review and rate products. See [a Backcountry.com product page](#) for an example.
- **Product QnA:** functionality allow customers to ask and answer questions on products. See [a Backcountry.com product page](#) for an example.
- **Product SEO (URL, title, meta data control):** functionality to allow site administrators to manage product URLs, product page titles, and product meta data.
- **Advanced/flexible taxonomy:** functionality to build a custom taxonomy / navigation structure for product browsing. For example, build multiple categories and subcategories with complex hierarchy. The taxonomy at [Spree's demo](#) includes two categories of brand and category and subcategories in each.
- **SEO for taxonomy pages:** functionality to allow site administrators to manage taxonomy URLs, taxonomy page titles, and taxonomy meta data.
- **Configurable product search:** functionality to allow the developers and site administrators to adjust parameters used in search, such as products to show per page, and to show products with no on hand stock.
- **Bundled products for discount:** functionality to allow site administrators to create bundles or group products together and then apply a discount to the entire bundle. For example, product X, Y and Z purchased together will yield a \$10.00 discount.
- **Recently viewed products:** functionality to show customers products they recently visited. This can be displayed on other product pages or the navigation pages to aid in navigation back to those products if the user would like to revisit the products. See the image below for an example of this functionality in action at [Paper Source](#).



- **Soft product support/downloads:** functionality to sell soft products such as mp3 or pdf files.
- **Product comparison:** functionality to allow customers to compare multiple products, such as a comparison of price or technical features. See [Backcountry.com](#) for an example.
- **Upsell:** functionality to encourage the customer to purchase a similar product from a higher price point, or to purchase an add on, may or may not include the functionality to allow site administrators to manage the upsell products.
- **Cross sell:** functionality to encourage the customer to purchase related items, may or may not include the functionality to allow site administrators to manage the cross sell products. See the image below for an example of this functionality at [Backcountry.com](#) after adding an item to the cart.

### Shopping Cart



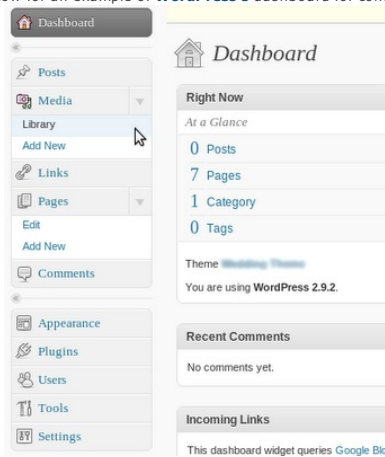
- **Related items:** functionality to display related items on product pages, may or may not include the functionality to allow site administrators to manage the related products.
- **RSS feed of products:** functionality to produce a RSS feed with product detail releases. [steepandcheap.com](#) offers a RSS feed, however, they are unique in that they offer a one deal at a time business model, so the RSS feed contains a stream of products for sale. This may or may not include the functionality to allow site administrators to manage the RSS feed contents.
- **Multiple images per product:** functionality to allow site administrators to upload multiple products per image. See [a Spree demo product](#) for an example.
- **Product option selection (variants):** functionality to allow site administrators to create and manage variants for



products to offer multiple variants per product, such as variants by size and color. See [a Spree demo product](#) for an example.

- **Wish list:** functionality to allow customers to create product wish lists. See [Amazon.com](#) for a description of their wish list functionality.
- **Send product email to friend:** functionality to allow customers to send emails to their friend to visit a specific product.
- **Product tagging / search by tagging:** functionality to allow site administrators to assign tags to products for navigation or searching. See [CityPass's blog](#) for an example of tag use in a content management system; in ecommerce context, the tags would navigate to a set of products instead of a set of blog articles.
- **Breadcrumbs:** functionality that renders the product navigation hierarchy on navigation and product pages to allow customers to navigate to previous pages visited. See [a Spree demo navigation page](#) for an example.

- **Blogging functionality:** functionality to allow site administrators to create, manage and display blog articles.
- **Static page management:** functionality to allow site administrators to create, manage, and display static pages such as "About Us", "Information".
- **Media management:** functionality to allow site administrators to create, manage, and display media such as images, video, audio. See the image below for an example of **WordPress's** dashboard for content management.



- **Contact us form:** functionality to allow customers to submit a request for contact. See [End Point's contact page](#) for an example.
- **Polls:** functionality to allow site administrators to create and manage basic polls, functionality to allow customers to submit answers to basic polls.

- **One page checkout:** functionality to allow customers to complete checkout on one page, rather than move forward through checkout through multiple address, payment pages. See [Paper Source](#) or [Backcountry.com's](#) checkout processes for examples.
- **Guest checkout:** functionality to checkout without creating a user account. Checkout at the [Spree demo](#) without being logged in for an example.
- **SSL support:** functionality to configure use of SSL during checkout. In Spree's case, the site administrator may turn SSL off during development and on during production.
- **Discounts:** functionality to allow customers to apply discount coupons to orders for a percentage or dollar amount reduction.
- **Gift certificates:** functionality to allow customers to purchase and use gift certificates as credit for purchases.
- **Saved shopping cart:** the functionality to save the products in a customers shopping cart so their shopping cart will be pre-populated on their next visit.
- **Saved addresses:** functionality to allow customers to create and manage addresses to be selected during checkout for billing or shipping rather than requiring the customer to re-enter their address. See the image below for an example of using saved addresses during [Backcountry.com's](#) checkout.

2 Verify Your Billing And Shipping Address

BILLING ADDRESS:	SHIPPING ADDRESS:
Test Tester 12340 W. Carolina Dr. Lakewood, CO 80228	Test Tester2 12340 W. Carolina Dr. Lakewood, CO 80228
<a href="#">Edit</a> <a href="#">Add New</a> or Choose from Billing Address Book:	<a href="#">Edit</a> <a href="#">Add New</a> or Choose from Shipping Address Book:
Test	Test2

3 Choose A Shipping Option

Your order qualifies for FREE shipping! Select the Economy (7-14 Days) shipping option below.

Select USPS if shipping to a PO Box.

Shipping Method	Cost
Standard (5-7 days)	\$6.16
Economy (7-14 days)	FREE!
3 Day Shipping	\$11.18

- **Real time rate lookup (UPS, USPS, FedEx):** the functionality to request rates from UPS, USPS, or FedEx during checkout for more accurate rate pricing rather than using a flat shipping rate.
- **Order tracking:** functionality to allow the site administrators to enter order tracking information and allow the user to review that tracking information, may or may not include sharing this information in a "Your order has been shipped" email.
- **Multiple shipments per order:** functionality to allow site administrators to split orders into multiple packages if specific products can not be shipped at the same time or can not be shipped together. See the image below for an example of [Spree's](#) backend shipping interface.

Order R524074868

Shipment #	Shipping Method	Cost	Tracking	Status	Date/Time
R524074868	UPS Ground	\$5.00		Pending	

Continue

- **Complex rate lookup:** functionality to calculate ship rates based on weight or price.
- **Free shipping:** functionality to offer free shipping.

- **Multiple payment gateways:** integration of multiple payment gateways such as Authorize.NET, Beanstream, Paypal, SagePay, etc.
- **Authorize.Net:** integration of the Authorize.Net payment gateway; may or may not include the use of profiles (Authorize.Net CIM).

- **Authorize and capture versus authorize only:** functionality to allow site administrators to configure whether or not credit cards should be authorized only during checkout completion or authorized and captured. If the credit card is authorized only, site administrators may finalize an order by capturing on the backend interface.
- **Google Checkout:** integration of **Google Checkout**.
- **Paypal Express:** integration of **Paypal Express**.

- **Sales reporting:** the functionality to display sales statistics, such as profits on sales or year-over-year sales.
- **Sales management tools:** functionality to allow site administrators to create and manage product sales. For example, the site administrator might create a 50% off sale to cover 25% of the products to begin in a week and end in two weeks.
- **Inventory management:** functionality to allow the site administrator to manage individual inventory units and their current state (on\_hand, shipped, backordered) and order assignment.
- **Purchase order management:** functionality to allow the site administrator to create and manage purchase orders. See the image below for an example of a potential backend interface for purchase orders in Spree.

#### Purchase Orders

New Purchase Order					
Name	Comments	Date Placed	Expected Fulfillment Date	Fulfillment Date	Action
Purchase Order 1	Sample purchase order data comment 1	2010-07-09	2010-07-10		<a href="#">Edit</a> / <a href="#">Delete</a>

- **Multi-tier pricing for quantity discounts:** functionality to allow customers to buy large quantities of products at a discount, the functionality to allow the site administrator to manage the large quantity product discounts.
- **Landing page tool:** functionality to create custom landing pages that may include targeted content or products, typically used for advertising or marketing.
- **Batch import and export of products:** functionality to allow the site administrator to import and export products via admin interface or script rather than entering each product individually.
- **Multiple sales reports:** See "Sales reporting" above.
- **Order fulfillment:** functionality to allow site administrators to manage fulfillment (inventory selection, shipping) of orders.
- **Tax Rate Management:** functionality to manage tax rates per zone, where zones are defined by states and/or countries. Note that in Spree, zones can only be defined by a combination of states and countries and tax rates can be tied to one or more zones.

- **User addresses:** See "Saved addresses" above.
- **Feature rich user preferences:** integration of various user account tools, such as address management, profile management, order review, etc.
- **Order tracking history:** functionality to allow a customer to lookup their order history, may or may not include order tracking information.

- **Extensibility:** functionality to extend the ecommerce core with modular components.
- **Appearance Theming:** functionality to change the appearance of the site.
- **Ability to customize appearance at category or browsing level:** functionality to create and manage custom and varied appearances for product browsing pages. For example, the categories "Jackets" and "Pants" may have different appearances, motivated by marketing or advertising.
- **Localization:** the functionality to translate the ecommerce site to a different language. See the image below for a small example of localization in action in Spree.



- **Multi-store, single admin support:** functionality to manage multiple stores from a single administrative location. An example of this might include <http://store1.endpoint.com/>, and <http://store2.endpoint.com/>, where both can be managed at <http://admin.endpoint.com/>.
- **Support for multiple currencies:** functionality to translate product prices between currencies.
- **Web service API:** functionality to retrieve data from the ecommerce application for third party use. See **Spree's documentation** on the Spree API.
- **System wide SEO:** general site-wide SEO functionality including features such as sitemap, googlebase integration, URL management, page title management.
- **Google Analytics:** functionality to allow site administrators to create and manage Google Analytics Ids, functionality to track traffic and conversion on the frontend.
- **Active community:** an active developer community with frequent core and extension contributions.

Most of the features described above are well known to ecommerce developers, but this list might also serve as a good checklist to review with a potential client during the estimate process to make sure expectations of an ecommerce platform are managed, especially with a young platform such as Spree where some features are not yet included in the core.

Learn more about End Point's **Ecommerce Development** or **Ruby on Rails Ecommerce Services**.

## Upgrading Spree with the help of Git

Lately, I've upgraded a few Spree projects with the recent Spree releases. Spree is a Ruby on Rails ecommerce platform that End Point previously sponsored and continues to support. In all cases, my Spree project was running from the Spree gem (version 0.10.2) and I was upgrading to Spree 0.11.0. I wanted to go through a brief explanation on how I went about upgrading my projects.



First, I made sure my application was running and committed all recent changes to have a clean branch. I follow the development principles outlined **here** that describe methodology for developing custom functionality on top of the Spree framework core. All of my custom functionality lives in the `RAILS_ROOT/vendor/extensions/site/` directory, so that directory probably won't be touched during the upgrade.

```
steph@The-Laptop:~/var/www/ep/myproject$ git status
# On branch master
nothing to commit (working directory clean)
```

Then, I tried the rake spree:upgrade task with the following results. I haven't upgraded Spree recently, and I vaguely remembered there being an upgrade task.

```
steph@The-Laptop:~/var/www/ep/myproject$ rake spree:upgrade
(in /var/www/ep/myproject)
[find_by_param error] database not available?
```

This task has been deprecated. Run 'spree --update' command using the newest gem instead.

OK. The upgrade task has been removed. So, I try spree --update:

```
Updating to Spree 0.11.0 ...
Finished.
```

That was easy! I run 'git status' and saw that there were several modified config/ files, and a few new config/ files:

```
# On branch master
# Changed but not updated:
#   (use "git add ..." to update what will be committed)
#   (use "git checkout -- ..." to discard changes in working directory)
#
#       modified:   config/boot.rb
#       modified:   config/environment.rb
#       modified:   config/environments/production.rb
#       modified:   config/environments/staging.rb
#       modified:   config/environments/test.rb
#       modified:   config/initializers/locales.rb
#       modified:   config/initializers/new_rails_defaults.rb
#       modified:   config/initializers/spree.rb
#
# Untracked files:
#   (use "git add ..." to include in what will be committed)
#
#       config/boot.rb~
#       config/environment.rb~
#       config/environments/cucumber.rb
#       config/environments/production.rb~
#       config/environments/staging.rb~
#       config/environments/test.rb~
#       config/initializers/cookie_verification_secret.rb
#       config/initializers/locales.rb~
#       config/initializers/new_rails_defaults.rb~
#       config/initializers/spree.rb~
#       config/initializers/touch.rb
#       config/initializers/workarounds_for_ruby19.rb
```

Because I had a clean master branch before the upgrade, I can easily examine the code changes for this upgrade from Spree 0.10.2 to Spree 0.11.0:

config/boot.rb

```
- load_rails("2.3.5") # note: spree requires this specific version of rails (change at your own risk)
+ load_rails("2.3.8") # note: spree requires this specific version of rails (change at your own risk)
```

config/environment.rb

```
- config.gem 'authlogic', :version => '>=2.1.2'
+ config.gem 'authlogic', :version => '2.1.3'
- config.gem 'will_paginate', :lib => 'will_paginate', :version => '2.3.11'
+ config.gem 'will_paginate', :lib => 'will_paginate', :version => '2.3.14'
- config.i18n.default_locale = :en-US
+ config.i18n.default_locale = :en'
```

config/environments/test.rb

```
-config.gem 'test-unit', :lib => 'test/unit', :version => '>2.0.5' if RUBY_VERSION.to_f >= 1.9
+config.gem 'test-unit', :lib => 'test/unit', :version => '>2.0.9' if RUBY_VERSION.to_f >= 1.9
```

Most of the changes were not surprising, except that the locale changes here are significant because they may require extension locales to be updated. After I reviewed these changes, I installed newer gem dependencies and bootstrapped the data since all my application data was stored in sample data files and restarted the server to test the upgrade. Then, I added the config/ and public/ files in a single git commit. I removed the old temporary configuration files that were left around from the upgrade.

For this particular upgrade, my git log shows changes in the files below. The config/ files were made when I ran the update, and the public/ files were modified when I restarted the server as gem public/ files are copied over during a restart.

```
commit 96a68e86064aa29f51c5052631f896845c11c266
Author: Steph Powell
Date: Mon Jun 28 13:44:50 2010 -0600

    Spree upgrade.

diff --git a/config/boot.rb b/config/boot.rb
diff --git a/config/environment.rb b/config/environment.rb
diff --git a/config/environments/cucumber.rb b/config/environments/cucumber.rb
diff --git a/config/environments/production.rb b/config/environments/production.rb
diff --git a/config/environments/staging.rb b/config/environments/staging.rb
diff --git a/config/environments/test.rb b/config/environments/test.rb
diff --git a/config/initializers/cookie_verification_secret.rb b/config/initializers/cookie_verification_secret.rb
diff --git a/config/initializers/locales.rb b/config/initializers/locales.rb
diff --git a/config/initializers/new_rails_defaults.rb b/config/initializers/new_rails_defaults.rb
diff --git a/config/initializers/spree.rb b/config/initializers/spree.rb
diff --git a/config/initializers/touch.rb b/config/initializers/touch.rb
diff --git a/config/initializers/workarounds_for_ruby19.rb b/config/initializers/workarounds_for_ruby19.rb
diff --git a/public/images/admin/bg/spree_50.png b/public/images/admin/bg/spree_50.png
Binary files a/public/images/admin/bg/spree_50.png and b/public/images/admin/bg/spree_50.png differ
diff --git a/public/images/tile-header.png b/public/images/tile-header.png
Binary files /dev/null and b/public/images/tile-header.png differ
diff --git a/public/images/tile-slider.png b/public/images/tile-slider.png
Binary files /dev/null and b/public/images/tile-slider.png differ
diff --git a/public/javascripts/admin/checkouts/edit.js b/public/javascripts/admin/checkouts/edit.js
diff --git a/public/javascripts/taxonomy.js b/public/javascripts/taxonomy.js
diff --git a/public/stylesheets/admin/admin-tables.css b/public/stylesheets/admin/admin-tables.css
diff --git a/public/stylesheets/admin/admin.css b/public/stylesheets/admin/admin.css
diff --git a/public/stylesheets/screen.css b/public/stylesheets/screen.css
```

From my experience, the config/ and public/ files are typically modified with a small release. If your project has custom JavaScript, or overrides the Spree core JavaScript, you may have to review the upgrade changes more carefully. Version control goes a long way in highlighting changes and problem areas. Additionally, having custom code abstracted from the Spree core should allow for easier maintenance of your project.

Learn more about End Point's [Ecommerce Development](#) or [Ruby on Rails Ecommerce Services](#).

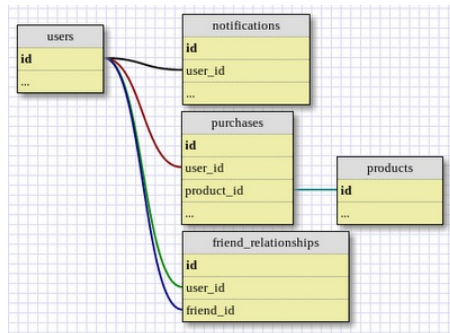
## NoSQL at RailsConf 2010: An Ecommerce Example

Even more so than Rails 3, **NoSQL** was a popular technical topic at RailsConf this year. I haven't had much exposure to NoSQL except for reading a few articles written by [Ethan \(Quick Thoughts on NoSQL Live Boston Conference, NoSQL Live: The Dynamo Derivatives \(Cassandra, Voldemort, Riak\), and Cassandra, Thrift, and Fibers in EventMachine\)](#), so I attended a few sessions to learn more.

First, it was reinforced several times that if you can read JSON, you should have no problem comprehending NoSQL. So, it shouldn't be too hard to jump into code examples! Next, I found it helpful when one of the speakers presented high-level categorization of NoSQL, whether or not the categories meant much to me at the time:

- **Key-Value Stores:** Advantages include that this is the simplest possible data model. Disadvantages include that range queries are not straightforward and modeling can get complicated. Examples include Redis, Riak, Voldemort, Tokyo Cabinet, MemcacheDB.
- **Document stores:** Advantages include that the value associated with a key is a document that exposes a structure that allows some database operations to be performed on it. Examples include CouchDB, MongoDB, Riak, FleetDB.
- **Column-based stores:** Examples include Cassandra, HBase.
- **Graph stores:** Advantages include that this allows for deep relationships. Examples include Neo4j, HypergraphDB, InfoGrid.

In one NoSQL talk, **Flip Sasser** presented an example to demonstrate how an ecommerce application might be migrated to use NoSQL, which was the most efficient (and very familiar) way for me to gain an understanding of NoSQL use in a Rails application. Flip introduced the models and relationships shown here:



In the transition to NoSQL, the transaction model stays as is. As a purchase is created, the Notification.create method is called.

```
class Purchase < ActiveRecord::Base
  after_create :create_notification

  # model relationships
  # model validations

  def total
    quantity * product.price
  end

  protected
  def create_notification
    notifications.create({
      :action => "purchased #{quantity == 1 ? 'a' : quantity} #{quantity == 1 ? product.name : product.name.pluralize}",
      :description => "Spent a total of #{total}",
      :item => self,
      :user => user
    })
  end
end
```

Flip moves the product class to Document store because it needs a lot of flexibility to handle the diverse product metadata. The structure of the product class is defined in the product class and nowhere else.

#### Before

```
class Product < ActiveRecord::Base
  serialize :info, Hash
end
```

#### After

```
class Product
  include Mongomapper::Document

  key :name, String
  key :image_path, String

  key :info, Hash

  timestamps!
end
```

The Notification class is moved to a Key-Value store. After a user completes a purchase, the create method is called to store a notification against the user that is to receive the notification.

#### Before

```
class Notification < ActiveRecord::Base
  # model relationships
  # model validations
end
```

#### After

```
require 'ostruct'

class Notification < OpenStruct
  class << self
    def create(attributes)
      message = "#{attributes[:user].name} #{attributes[:action]}"
      attributes[:user].follower_ids.each do |follower_id|
        Red.lpush("user:#{follower_id}:notifications", {:message => message, :description => attributes[:description], :timestamp => Time.now}.to_json)
      end
    end
  end
end
```

The user model remains an ActiveRecord model and uses the devise gem for user authentication, but is modified to retrieve the notifications, now an OpenStruct. The result is that whenever a user's friend makes a purchase, the user is notified of the purchase. In this simple example, a purchase contains one product only.

#### Before

```
class User < ActiveRecord::Base
  # user authentication here
  # model relationships

  def notifications
    Notification.where("friend_relationships.friend_id = notifications.user_id OR notifications.user_id = #{id}")
    .joins("LEFT JOIN friend_relationships ON friend_relationships.user_id = #{id}")
  end
end
```

#### After

```
class User < ActiveRecord::Base
  # user authentication here
  # model relationships

  def followers
    User.where('users.id IN (friend_relationships.user_id)').
      joins("JOIN friend_relationships ON friend_relationships.friend_id = #{id}")
  end

  def follower_ids
    followers.map(&:id)
  end

  def notifications
    (Red.lrange("user:#{id}:notifications", 0, -1) || []).map{|notification| Notification.new(ActiveSupport::JSON.decode(notification))}
  end
end
```

The disadvantages to the NoSQL and RDBMS hybrid is that data portability is limited and ActiveRecord plugins can no longer be used. But the general idea is that performance justifies the move to NoSQL for some data. In several sessions I attended, the speakers reiterated that you will likely never be in a situation where you'll only use NoSQL, but that it's another tool available to suit performance-related business needs. I later spoke with a few **Spree** developers and we concluded that the NoSQL approach may work well in **some** applications for product and variant data for improved performance with flexibility, but we didn't come to an agreement on where else this approach may be applied.

Learn more about End Point's [Ruby on Rails Development](#) or [Ruby on Rails Ecommerce Services](#).

## Rails 3 at RailsConf 2010: Code Goodness

At RailsConf 2010, popular technical topics this year are Rails 3 and NoSQL technologies. My first two articles on RailsConf 2010 so far ([here](#) and [here](#)) have been less technical, so I wanted to cover some technical aspects of Rails 3 and some tasty code goodness in standard ecommerce examples.

**Bundler**, a gem management tool, is a hot topic at the conference, which comes with Rails 3. I went to a talk on Bundler and it was mentioned in several talks, but a quick run through on its use is:

```
gem install bundler
gem update --system # update Rubygems to 1.3.6+
```

Specify your gem requirements in the application root Gemfile directory.

```
# excerpt from Spree Gemfile in the works
gem 'searchlogic',      '2.3.5'
gem 'will_paginate',    '2.3.11'
gem 'faker',            '0.3.1'
gem 'paperclip',        '>=2.3.1.1'

bundle install # installs all required gems
git add Gemfile # add Gemfile to repository
```

In Spree, the long-term plan is to break apart ecommerce functional components into gems and implement Bundler to aggregate the necessary ecommerce gems. The short-term plan is to use Bundler for management of all the Spree gem dependencies.

ActiveRecord has some changes that affect the query interface. Some ecommerce examples on new querying techniques with the idea of chaining finder methods:

```
recent_high_value_orders = Order
  .where("total > 1000")
  .where(["created_at >= :start_date", { :start_date => params[:start_date] }])
  .order("created_at DESC")
  .limit(50)
```

An example with the use of scope:

```
class Order << ActiveRecord::Base
  scope :high_value_orders where("total > 1000")
    .where(["created_at >= :start_date", { :start_date => Time.now - 5.days }])
    .order("created_at DESC")
end

class SomeController << YourApplication::AdminController
  def index
    orders = Order.high_value_orders.limit(50)
  end

  def snapshot
    orders = Order.high_value_orders.limit(10)
  end

  def winner
    Order.high_value_orders.first
  end
end
```

The changes to ActiveRecord provide a more sensible and elegant way to build queries and moves away from the so-called drunkenness on hashes in Rails. ActiveRecord finder methods in Rails 3 include where, having, select, group, order, list, offset, joins, includes, lock, read only, and from. Because the relations are lazily loaded, you have the ability to chain query conditions with no performance effects as the query hasn't been executed yet, and fragment caching is more effective because the query is executed from a view call. Eager loading can be forced by using first, last, and all.

Some new changes are introduced with Rails 3 in routing that move away from hash-itis, clarify flow ownership, and improve conceptual conciseness. A new route in a standard ecommerce site may be:

```
resources :users do
  member do
    get :index, :show
  end
  resources :addresses
  resources :reviews
  post :create, :on => :member
end
```

Another routing change on named routes allows:

```
get 'login' => 'sessions#new' # sessions is the controller, new is the action
```

Some significant changes were changed to the ActionMailer class after a reexamination of assumptions and the decision to model mailers after a Rails controller instead of a model/controller hybrid. An example of use with ActionMailer now:

```
class OrderCompleteNotifier < ActionMailer::Base
  default :from => "customerservice@myecommercesite.com"

  def order_complete_notification(recipient)
    @recipient = recipient
    mail(:to => recipient.email_address_with_name,
        :subject => "Order information here")
  end
end
```

And some changes in sending messages, allowing the following:

```
OrderCompleteNotifier.signup_notification(recipient1).deliver # sends email
message = OrderCompleteNotifier.signup_notification(recipient2)
message.deliver
```

A few talks about Rails 3 mentioned the use of **RailTies**, which serves as the interface between the Rails framework and the rest of its components. It accepts configuration from application.rb, sets up initializers in extensions, tells Rails about generators and rake tasks in extensions, gems, plugins.

**DHH** briefly spoke about some Rails 3.1 things he's excited about, including reorganization of the public directory assets and implementing sprite functionality, which I am a big fan of.

A few recommended Rails 3 learning resources were mentioned throughout the conference, including:

- **Rails 3 Screencasts** by Gregg Pollack
- **The Rails 3 Upgrade Handbook** by Jeremy McAnally
- **The Rails Dispatch Blog**
- **The Great Decoupling** by Yehuda Katz
- **Rails API Documentation**

There are tons of resources out there on these topics and more that I found as I was putting this article together. Go look and write code!

Learn more about End Point's **Ruby on Rails Development** or **Ruby on Rails Ecommerce Services**.

## RailsConf 2010: Spree and The Ecommerce Smackdown, Or Not

Spree has made a good showing at RailsConf 2010 so far this year, new site and logo released this week:

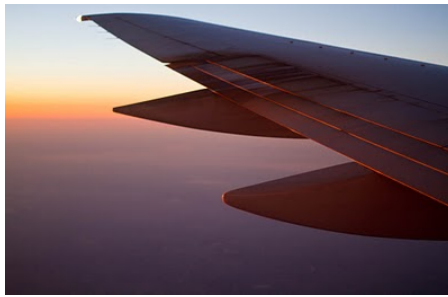


It started off in yesterday's Ecommerce Panel with Sean Schofield, a former End Point employee and technical lead on Spree, Cody Fauser, the CTO of **Shopify** and technical lead of **ActiveMerchant**, Nathaniel Talbott, co-founder of **Spredly**, and Michael Bryzek, the CTO and founder of **Gilt Groupe**.

The panel gave a nice overview on a few standard ecommerce questions:

- **My client needs a store - what technology should I use? Why shouldn't I just reinvent the wheel?** The **SaaS** reps evangelized their technologies well, explaining that a hosted solution is good as a relatively immediate solution that has minimum cost and risk to a business. A client [of SaaS] need not be concerned with infrastructure or transaction management initially, and a hosted solution comes with *freebies* like the use of a CDN that improve performance. A hosted solution is a good option to get the product out and make money upfront. Also, both SaaS options offer elegant APIs.
- **How do you address client concerns with security?** Again, the SaaS guys stressed that there are a few mistakes guaranteed to kill your business and one of those things includes dropping the ball on security, specifically credit card security. The hosted solutions worry about credit card security so the client doesn't have to. One approach to PCI compliance is to securely post credit card information to a 3rd party secure payment request API as the external requests minimizes the risk to a company. Michael (Gilt) discussed The Gilt Groupe's intricate process in place for security and managing encryption keys. Nathaniel (Spredly) summarized that rather than focus on PCI compliance specifically, it's more important to have the right mindset about financial data security.
- **What types of hosting issues should I be concerned about?** Next, the SaaS guys led again on this topic by explaining that they worry about the hosting - that a monthly hosted solution cost (Shopify.com starts at \$24/month) is less than the cost of paying a developer who knows your technology in an emergency situation when your site goes down on subpar hosting. Michael (Gilt) made a good point by considering that everything is guaranteed to fail at some point - how do you (client) feel about taking the risk of that? do you just trust that the gateway is always up? One interesting thing mentioned by the SaaS guys is that technically, you should not be able to host any solution in the cloud if you touch credit card data, although you may likely be able to "get away with it" - I'm not sure if this was a scare tactic, but it's certainly something to consider. One disadvantage to hosting in the cloud are that you can't do forensic investigation after a problem if the machine has disappeared.

The remaining panel time was spent on user questions that focused on payment and transaction details specifically. There were a few bits of valuable transaction management details covered. Cody (Shopify) has no plans of developing or expanding on alternative payment systems because there isn't a good ROI. The concept of having something like OIDs for credit cards to shop online would likely not receive support from credit card companies, but PayPal [kinda] serves this role currently. Nathaniel (Spredly) covered interesting details on how user transaction information is tracked: From day one, everything that is stacked on a users account is a transaction model object that mutate the user transaction state over time. The consensus was that a transaction log is the way to track user transaction information - you should never lose track of any dollarz. On the topic of data, Shopify and Spredly collect and store all data - the first client step is to sell stuff, then later the client can come back to analyze data for business intelligence such as ROI per customer demographic, the average lifespan of a customer, or the computed value of a customer.



Now I take a break for an image, because it's important to have images in blog articles. Here is my view from the plane as I traveled to Baltimore.

After the panel, Spree had a Birds of a Feather session in the evening, which focused more on Spree. Some topics covered:

- **What is the current Spree road map to work on Rails 3? Extension development in Rails 3?** As Rails 3 stabilizes over time, Spree will begin to transition but no one's actively doing Rails 3 work at this point. I spoke with Brian Quinn, a member of the Spree core team, who mentioned that he's recently spent time on investigating **resource controller** versus inherited resources or something else. The consensus was that people don't like Resource Controller (one attendee mentioned they used the Spree data model, but ripped out all of the controllers), but that a sensible alternative will need to be implemented at some point. **Searchlogic**, the search gem used in Spree search functionality, has no plans to upgrade to Rails 3, so Spree will also have to make sensible decisions for search. The Rails 3 generators have a reputation to be good, so this may trickle down to have positive effects on Spree extension development. Rails 3 also encourages more modular development, so the idea with Spree is that things will gradually be broken into more modular pieces, or gems, and the use of **Bundler** will tie the Spree base components together.
- **How's test coverage in Spree?** Bad. Contributors appreciated.
- **I got scared after I went to the Ecommerce Panel talk - what's PCI compliance and security like in Spree?** Spree is PCI compliant with the assumption that the client doesn't store credit cards - there is (was?) actually a Spree preference setting that defaults to not store credit cards in the database, but it will result in unencrypted credit cards being stored in the database if set to true. The Spree core team **recently mentioned** that this might be removed from the core. Offsite credit card use such as Authorize.Net CIM implementation is included in the Spree core.

The Spree Birds of a Feather session was good: the result of the session was likely a comprehension of the short and long term road map of Spree as it transitions to Rails 3. This blog post was going to end here, but luckily I sat next to a Shopify employee this morning and learned more about Shopify. My personal opinion of the ecommerce panel was that advantages of the hosted solutions were appropriately represented, but there wasn't much focus on Spree and the disadvantages of SaaS weren't covered much. I learned that one major disadvantage to Shopify is that they don't have user accounts, however, user accounts are in development. Shopify is also [obviously] not a good choice for sites that have customization but there is a large community of applications. One example of a familiar customization is building a site with a one-deal-at-a-time business model (**SteepAndCheap.com**, **JackThreads.com's** former business model) - this would be difficult with Shopify. Some highlights of Shopify include it's template engine, based on **Liquid**, it has a great API, where you can do most things except place an order, and that it scales really well.

Obviously, I **drink the Spree kool-aid often**, so I learned more from the panel, BOF, and hallway talk on the subject of SaaS, or hosted Rails ecommerce solutions. The BOF session covered details on the Spree to Rails 3 (hot topic!) transition nicely.

Learn more about End Point's **Ruby on Rails Development** or **Ruby on Rails Ecommerce Services**.

## RailsConf 2010 Rate a Rails Application: Day One, Session One

My first session at RailsConf 2010 was one that I found valuable: *12 hours to Rate a Rails Application* presented by Elise Huard. Elise discussed the process of picking up a Rails application and analyzing it, rather than being the developer who develops a Rails application from scratch. This is particularly valuable because I've had to dive into and comprehend Rails projects more in the last few months. I'd imagine this will become more common as Rails matures and legacy code piles up. Elise mentioned that her motivation for application review comes from either acquisition or for project maintenance. Below is the 12-hour timeline covered by Elise:

First, Elise suggests that speaking to a team will reveal much about the application you are about to jump into. In our case at End Point, we often make up part of or all of the development team. She briefly mentioned some essential personality traits to look for:

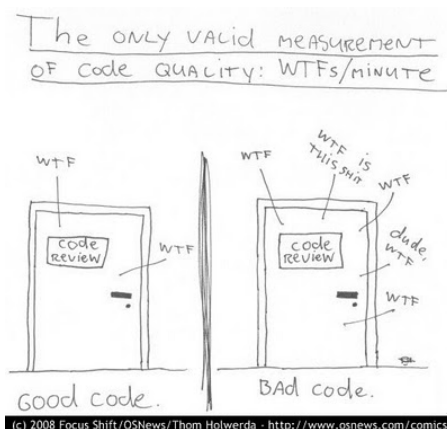
- control freak: someone who worries about details and makes other people pay attention to details
- innovator: someone who looks for the next exciting things and doesn't hesitate to jump in
- automator: people who care about process, more sys admin side of things
- visionaries
- methodologizers (ok, i made this word up): someone who has long term planning ability, road mapping insight
- humility: important to be open to understanding code flaws and improve

Of course, there's overlap of personality traits, but the point is that these traits are reflected in the code base in some way or another. Elise briefly mentioned that having an issue tracker or version control is viewed positively in application review (of course).

The next step in a Rails application evaluation is running the app, making sure it works, examining the maintainability, rails version, and license. She also discussed avoiding the **NIH syndrome** during a review, which I interpreted as reviewing the code and avoiding thinking about reinventing of the wheel and taking the code and functionality as is (not sure I interpreted her intentions correctly) rather than immediately deciding that you would rewrite everything. Additional systemic indications of a good application are applications that use open source gems or plugins that are maintained and used by others, and that the application has passing tests.

The next step in a 12-hour Rails application review should be an initial poke around of the code. Elise likes to look at `config/routes.rb` because it's an interface application to the user and a good `config/routes.rb` file will be a representative inventory of the application. Another step in the review is to examine a model diagram, using a tool such as the railroad gem, or via rubymine. Another good overview is to examine how parts of the application are named, as the names should be understandable to someone in the business.

Elise's next step in application review is using several metrics to examine complexity and elegance of code, which covered several tools that I haven't heard of besides the common and popular (already mentioned a few times at RailsConf) WTF-metric.



An overview of the tools:

- "rake stats": lines of codes, methods, etc. per controllers, helpers, models
- **parsetree's ruby\_parser**: code transformed into an abstract syntax tree
- flog: analysis of code complexity based on the ABC (Assignment Branch Condition) metric
- flay: analysis for similarity between classes
- **saikuro**: analysis of **cyclomatic complexity**
- **roodi**: detection of antipatterns using the visitor pattern
- reek: detection OO code smells
- rails\_best\_practices: smell for rails antipatterns
- churn: metrics on change of code run on version control history
- rcov: analysis of code coverage
- **metric\_fu**: tool aggregate that includes many of above tools

Elise noted that although metrics are valuable, they don't identify bugs, analyze code performance and don't analyze the human readability of the code.

Next up in an application review is looking at the good code. She likes to look at the database files: is everything in the migrations? is the database optimized sensibly? Occasionally, Rails developers can become a bit ignorant (sorry, true) to data modeling, so it's important to note the current state of the database. She also looks at the views to analyze style, look for divitis, and identify too much JavaScript or logic.

The next step in Elise's review of a Rails application is checking out the test code. As implementation or requirements change, tests should change. The tests should express code responsibility and hide the implementation detail. Tests should reveal expressive code and don't necessarily need to follow the development DRY standard.

Another point of review is to understand the deployment methodology. Automated deployment such as deployment using capistrano, chef, etc. are viewed positively. Similar to software development tests, failures should be expressive. Deployment is also viewed positively if performance tests and/or bottleneck identification is built into deployment practices.

In the final hour of application review, Elise looks for brownie-point-worthy coverage such as:

- continuous integration
- documentation and freshness of documentation
- monitoring (nagios, etc.), exception notification, log analyzers
- testing javascript

I found this talk to be informative on how one might approach understanding an existing rails application. As a consultant, we frequently have to pick up a project and **just go**, Rails or not, so I found the tools and approach presented by Elise insightful, even if I might rearrange some of the tasks if I am going to write code.

The talk might also be helpful in providing details to teach someone where to look in an application for information. For example, a couple of End Point developers are starting to get into Rails, and from this talk I think it's a great recommendation to send someone to config/routes.rb to learn and understand the application routing as a starting point.

Learn more about End Point's [Ruby on Rails Development](#) or [Ruby on Rails Ecommerce Services](#).

## Spree vs Magento: A Feature List Comparison

This week, a client asked me for a list of Spree features both in the core and in available extensions. I decided that this might be a good time to look through Spree and provide a comprehensive look at features included in Spree core and extensions and use Magento as a basis for comparison. I've divided these features into meaningful broader groups that will hopefully ease the pain of comprehending an extremely long list :) Note that the Magento feature list is based on their documentation. Also note that the Spree features listed here are based on recent 0.10.\* releases of Spree.

Feature	Spree	Magento
Product reviews and/or ratings	Y, extension	Y
Product ana	N	N



Product seo (url, title, meta data control)	N	Y
Advanced/flexible taxonomy	Y, core	Y
Seo for taxonomy pages	N	Y
Configurable product search	Y, core	Y
Bundled products for discount	Y, extension	Y
Recently viewed products	Y, extension	Y
Soft product support/downloads	Y, extension	Y, I think so
Product comparison	Y, extension	Y
Upsell	N	Y
Cross sell	N	Y
Related items	Y, extension	Y
RSS feed of products	N	Y
Multiple images per product	Y, core	Y
Product option selection (variants)	Y, core	Y
Wishlist	Y, extension	Y
Send product email to friend	Y, extension	Y
Product tagging / search by tagging	N	Y
Breadcrumbs	Y, core	Y

Features	Spree	Magento
Blogging functionality	Y, extension	Y *extension
Static page management	Y, extension	Y
Media management	N	Y
Contact us form	Y, extension	Y
Polls	Y, extension	Y

Feature	Spree	Magento
One page checkout	N	Y
Guest checkout	Y, core	Y
SSL Support	Y, core	Y
Discounts	Y, core	Y
Gift Certificates	N	Y
Saved Shopping Cart	N	Y
Saved Addresses	Y, extension	Y

Feature	Spree	Magento
Real time rate lookup (UPS, USPS, Fedex)	Y, extension	Y
Order tracking	N	Y
Multiple shipments per order	Y, core	Y
Complex rate lookup	Y, extension	Y
Free shipping	Y, extension	Y

Feature	Spree	Magento
Multiple Payment Gateways	Y, core	Y
Authorize.net	Y, core	Y
Authorize and capture versus authorize only	Y, core	Y
Google Checkout	Y, extension	Y
Paypal Express	Y, extension	Y

Feature	Spree	Magento
Sales reporting	Y, core	Y
Sales Management Tools	N	Y
Inventory management	Y, core	Y
Purchase order management	N	Y
Multi-tier pricing for quantity discounts	N	Y
Landing page tool	Y, extension	Y
Batch import and export of products	Y, extension	Y
Multiple Sales reports	Y, core	Y
Order fulfillment	Y, core	Y
Tax Rate Management	Y, core	Y

Feature	Spree	Magento
User addresses	Y, extension	Y
Feature rich user preferences	N	Y
Order tracking history	Y, core	Y

Feature	Spree	Magento
Extensibility	Y, core	Y
Appearance Theming	Y, core	Y
Ability to customize appearance at category or browsing level	N	Y
Localization	Y, core	Y
Multi-store, single admin support	Y, extension	Y
Support for multiple currencies	N	Y
Web Service API	Y, core	Y
SEO System wide: sitemap, google base, etc	Y, extension	Y
Google Analytics	Y, core	Y
Active community	Y, N/A	Y

The configurability and complexity of each feature listed above varies. Just because a feature is provided within a platform does not guarantee that it will meet the desired business needs. Magento serves as a more comprehensive ecommerce platform out of the box, but the disadvantage may be that adding custom functionality may require more resources (read: more expensive). Spree serves as a simpler base that may encourage quicker (read: cheaper) customization development simply because it's in Rails and because the dynamic nature of Ruby allows for elegant extensibility in Spree, but a disadvantage to Spree could be that a site with a large amount of customization may not be able to take advantage of community-available extensions because they may not all play nice together.

Rather than focus on the platform features, the success of the development depends on the developer and his/her skillset. Most developers will say that **any** of the features listed above are doable in Magento, Spree, or Interchange (a Perl-based ecommerce platform that End Point supports) with an unlimited budget, but a developer needs to have an understanding of the platform to design a solution that is easily understood and well organized (to encourage

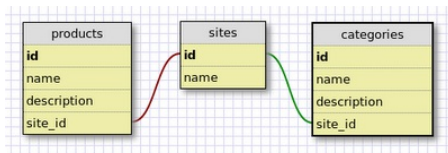
encapsulating the problem to design solution that is easy, maintainable and elegant (for example readability and understandability by other developers), develop with standard principles like DRY and MVC-style separation of concerns, and elegantly abstract from the ecommerce core to encourage maintainability. And of course, be able to understand the business needs and priorities to guide a project to success within the given budget. Inevitably, another developer will come along and need to understand the code and inevitably, the business will often use an ecommerce platform longer than planned so maintainability is important.

Please feel free to comment on any errors in the feature list. I'll be happy to correct any mistakes. Now, off to rest before RailsConf!

Learn more about End Point's [Ecommerce Development](#) or [Ruby on Rails Ecommerce Services](#).

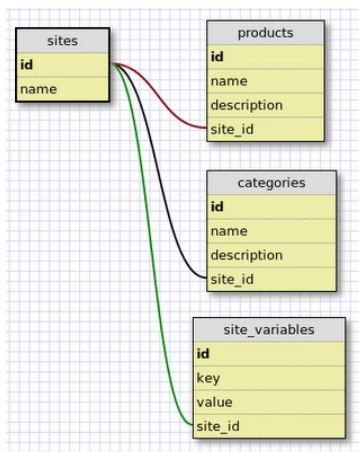
## Spree and Multi-site Architecture for Ecommerce

Running multiple stores from a single ecommerce platform instance seems to be quite popular these days. End Point has worked with several clients in developing a multi-store architecture running on one Interchange installation. In the case of our work with Backcountry.com, the data structure requires that a `site_id` column be included in product and navigation tables to specify which stores products and categories belong to. Frontend views are generated and "partial views" or "view components" are organized into a per-site directory structure and database calls request products against the current `site_id`. Below is a simplified view of the data structure used for Backcountry.com's multi-site architecture.



Basic multi-store data architecture

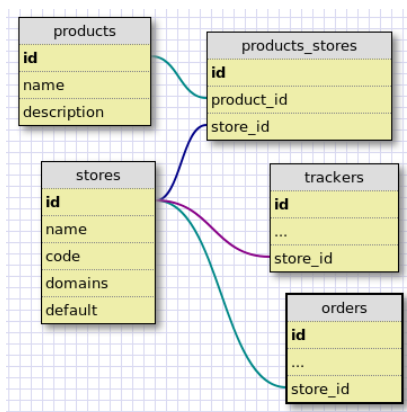
A similar data model was implemented and enhanced for another client, College District. A `site_id` column exists in multiple tables including the products and navigation tables, and sites can only access data in the current site schema. College District takes one step further in development for appearance management by storing CSS values in the database and enabling CSS stylesheet generation on the fly with the stored CSS settings. This architecture works well for College District because it allows the owners to quickly publish new sites. Below is a simplified view of the data structure used for College District, where the table `site_variables` contains CSS values (text, background colors, etc.).



Extended multi-store data architecture where `site_variables` table contains CSS settings. The store frontend can access data in the current store schema only.

In the past year, running a multi-site setup has been a popular topic in the user group for Spree, an open source Ruby on Rails platform. I've been happy to be involved in a couple of client multi-site projects. Here I'll discuss some comments for my Spree multi-site implementation work.

First, the [Spree multi-domain extension](#) developed by the Spree core team serves as a strong starting point. The extension migrations produce the data structure shown below, by creating a new table `stores` and `products_stores`, and adding a `store_id` column to the `trackers` (Google Analytics data) and `orders` table.



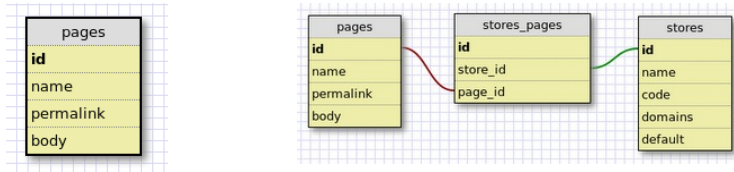
Code changes in addition to data model changes are:

- A before filter sets the current store by examining the current environment variable `"SERVER_NAME"`.
- Frontend product retrieval is modified to retrieve products in the current store only.

- Frontend navigation is generated based on products assigned to the current store.
- The Tracker (Google Analytics) retrieval method is modified to only retrieve the current store settings.
- Order processing assigns the store\_id value to each order.

With this extension, a user can check out across multiple stores running from a single Spree application with the same account, products can be assigned to multiple stores, and a single default store is set to render if no domains match the current SERVER\_NAME. The extension does not introduce the advanced schema behavior like College District's data architecture, however, the extension could be customized to do so. The extension suits basic requirements for a multi-store architecture.

In my project, there were additional changes required. I used the **Spree static pages extension**, which introduces functionality to present and manage static content pages. I modified this extension to create an additional stores\_pages table that introduces a has and belongs to many relationship between stores and pages.



Basic Spree static pages data model.

Expanded Spree static pages data model with multi-store design.

Other custom requirements may include modifying the **spree-faq extension** to build a has and belongs to many relationship between questions and stores, or similar changes that create relationships between the stores table and other data.

The next interesting design choice I faced was how to handle appearance management across multiple sites. As I mentioned before, a method was developed to retrieve and build views based on the current store in Backcountry.com's multi-store solution. With College District, the database stored CSS values and generated new stylesheets on the fly for each site. I chose to implement a simplified version of the College District implementation, where a single stylesheet contains the rules for each site.

In the Spree implementation, the Spree::BaseController class is modified with a before filter to set the store and asset (images, javascript, stylesheet) location:

```

module Spree::Fantebrate::BaseController
  def self.included(controller)
    controller.class_eval do
      controller.append_before_filter :set_store_and_asset_location
    end
  end
  def set_store_and_asset_location
    @current_store ||= Store.by_domain(request.env['SERVER_NAME']).first
    @current_store ||= Store.default.first
    @asset_location = @current_store.domains.gsub('mydomain.com', '')
  end
end

```

The default layout includes the main and site specific stylesheets:

```

<%= stylesheet_link_tag "style" %>
<%= stylesheet_link_tag "#{@asset_location}/site" %>

```

The site specific stylesheet contains style rules and includes site specific image settings:

```

body.site1 { background: #101a35 url(/images/site1/bg.jpg) repeat-x 0 0; }
.site1 h1#logo { float: left; display: inline; width: 376px; height: 131px; position:relative; left: -15px; padding-bottom: 10px; }
.site1 h1#logo a { display: block; height: 131px; background: url(/images/site1/logo.png); }
.site1 #top-right-info,
.site1 #top-right-info a,
.site1 #top-right-info b { color: #fff; }
...

```

This implementation acts on the assumption that there will be minimal design differences across stores. This is a simple and effective way to get an initial multi-store architecture in place that allows you to manage multiple site's appearance in a single Spree application.

Some advanced topics I've considered with this work are:

- Can we dynamically generate the migrations based on which extensions are installed? For example, a list of tables would be included in the main extension. This list of tables would be iterated through and if the table exists, a dynamic migration is generated to build a has many/belongs to, has one/belongs to, or has and belongs to many relationship between stores and the table.
- SSL setup for our Spree multi-store implementation was accomplished with a wildcard SSL certificate, where each store can be accessed from a different subdomain. Backcountry.com and College District implementation was accomplished with standard SSL certificates because the stores do not share domains. The Spree implementation methods described in this article do not vary for subdomain versus different domain design, but this is certainly something to consider at the start of a multi-site project.
- The multi-domain extension could be customized and enhanced to include a table that contains CSS settings similar to our College District implementation and allows you to generate new stylesheets dynamically to change the look of a site. The obvious advantage of this is that a user with site administrative permissions can change the appearance of the site via backend Spree management without involving development resources.

Learn more about End Point's **Ruby on Rails Development** or **Ruby on Rails Ecommerce Services**.

## Spree and Authorize.Net: Authorization and Capture Quick Tip

Last week I did a bit of reverse engineering on payment configuration in Spree. After I successfully setup Spree to use Authorize.net for a client, the client was unsure how to change the Authorize.Net settings to perform an authorize and capture of the credit card instead of an authorize only.

### Settings

Developer

Provider:	Gateway::AuthorizeNet
Test Mode:	<input checked="" type="checkbox"/>
Server:	test
Login:	
Password:	
<input type="button" value="Update"/>	

The requested settings for an Authorize.Net payment gateway on the Spree backend.

I researched in the Spree documentation for a bit and then sent out an email to the End Point team. **Mark Johnson** responded to my question on authorize versus authorize and capture that the Authorize.Net request type be changed from "AUTH\_ONLY" to "AUTH\_CAPTURE". So, my first stop was a grep of the activemerchant gem, which is responsible for handling the payment transactions in Spree. I found the following code in the gem source:

```
# Performs an authorization, which reserves the funds on the customer's credit card, but does not
# charge the card.
def authorize(money, creditcard, options = {})
  post = {}
  add_invoice(post, options)
  add_creditcard(post, creditcard)
  add_address(post, options)
  add_customer_data(post, options)
  add_duplicate_window(post)

  commit('AUTH_ONLY', money, post)
end

# Perform a purchase, which is essentially an authorization and capture in a single operation.
def purchase(money, creditcard, options = {})
  post = {}
  add_invoice(post, options)
  add_creditcard(post, creditcard)
  add_address(post, options)
  add_customer_data(post, options)
  add_duplicate_window(post)

  commit('AUTH_CAPTURE', money, post)
end
```

My next stop was the Spree payment\_gateway core extension. This extension is included as part of the Spree core. It acts as a layer between Spree and the payment gateway gem and can be swapped out if a different payment gateway gem is used without requiring changing the transaction logic in the Spree core. I searched for purchase and authorize in this extension and found the following:

```
def purchase(amount, payment)
  #combined Authorize and Capture that gets processed by the ActiveMerchant gateway as one single transaction.
  response = payment_gateway.purchase((amount * 100).round, self, gateway_options(payment))
  ...
end
def authorize(amount, payment)
  # ActiveMerchant is configured to use cents so we need to multiply order total by 100
  response = payment_gateway.authorize((amount * 100).round, self, gateway_options(payment))
  ...
end
```

My last stop was where I found the configuration setting I was looking for, Spree::Config[:auto\_capture], by searching for authorize and purchase in the Spree application code. I found the following logic in the Spree credit card model:

```
def process!(payment)
  begin
    if Spree::Config[:auto_capture]
      purchase(payment.amount.to_f, payment)
      payment.finalize!
    else
      authorize(payment.amount.to_f, payment)
    end
  end
end
```

The auto\_capture setting defaults to false, not surprisingly, so it can be updated with one of the following changes.

```
# * extension.rb:
def activate
  AppConfiguration.class_eval do
    preference :auto_capture, :boolean, :default => true
  end
end

# EXTENSION_DIR/config/initializers/*.rb:
if Preference.table_exists?
  Spree::Config.set(:auto_capture => true)
end
```

After I found what I was looking for, I googled "Spree auto\_capture" and found a few references to it and saw that it was briefly mentioned in the **Spree documentation payment information**. Perhaps more documentation could be added around how the Spree auto\_capture preference setting trickles down through the payment gateway processing logic, or perhaps this article provides a nice overview of the payment processing layers in Spree.

Learn more about End Point's **Ruby on Rails Development** or **Ruby on Rails Ecommerce Services**.

## Spree and Software Development: Git and Ruby techniques

Having tackled a few interesting Spree projects lately, I thought I'd share some software development tips I've picked up along the way.

### Gem or Source?

The first decision you may need to make is whether to run Spree from a gem or source. Directions for both are included at the **Spree Quickstart Guide**, but the guide doesn't touch on motivation from running from a gem versus source. The **Spree documentation** does address the question, but I wanted to comment based on recent experience. I've preferred to build an application running from the gem for most client projects. The only times I've decided to work against Spree source code was when the Spree edge code had a major change that wasn't available in a released gem, or if I wanted to troubleshoot the internals of Spree, such as the extension loader or localization functionality.

If you follow good code organization practices and develop modular and abstracted functionality, it should be quite easy to switch back and forth between gem and source. However, switching back and forth between Spree gem and source may not be cleanly managed from a version control perspective.

### git rebase

Git rebase is lovely. **Ethan** describes some examples of using git rebase [here](#). When working with several other developers and even when I'm the sole developer, I've included rebasing in my pull and push workflow.

## .gitmodules

Git submodules are lovely, also. An overview on git submodules with contributions from **Brian Miller** and **David Christensen** can be read [here](#). Below is an example of a .gitmodules from a recent project that includes several extensions written by folks in the Spree community:

```
[submodule "vendor/extensions/faq"]
  path = vendor/extensions/faq
  url = git://github.com/josnuuss/spree-faq.git
[submodule "vendor/extensions/multi_domain"]
  path = vendor/extensions/multi_domain
  url = git://github.com/railsdog/spree-multi-domain.git
[submodule "vendor/extensions/paypal_express"]
  path = vendor/extensions/paypal_express
  url = git://github.com/railsdog/spree-paypal-express.git
```

## .gitignore

This should apply to software development for other applications as well, but it's important to setup .gitignore correctly at the beginning of the project. I typically ignore database, log, and tmp files. Occasionally, I ignore some public asset files (stylesheets, javascripts, images) if they are copied over from an extension upon server restart, which is standard in Spree.

## Overriding modules, controllers, and views

Now, the good stuff! So, let's assume the Spree core is missing functionality that you need. Options for expanding from the Spree core include overriding or extending existing models, controllers, or views, or writing and including new models, controllers, or views. **Spree's Extension Tutorial** covers adding new controllers, models, and views, so I'll discuss extending and overriding existing models, views and controllers below.

To extend an existing controller, I've typically included a module with the extended behavior in the \*\_extension.rb file. For all examples, let's assume that my extension is named "Site", another standard in Spree. The code below shows the module include in site\_extension.rb:

```
...
def activate
  ProductsController.send(:include, Spree::Site::ProductsController)
end
...
```

My ProductsController module, inside the Spree::Site namespace, includes the following to define a before filter in the Spree core products controller:

```
module Spree::Site::ProductsController
  def self.included(controller)
    controller.class_eval do
      controller.append_before_filter :do_stuff
    end
  end

  def do_stuff
    #doing stuff
  end
end
```

Next, to override a method in an existing controller, I've started the same way as before, by including a module in site\_extension.rb:

```
...
def activate
  CheckoutsController.send(:include, Spree::Site::CheckoutsController)
end
...
```

The Spree::Site::CheckoutsController module will contain:

```
module Spree::Site::CheckoutsController
  def self.included(target)
    target.class_eval do
      alias :spree_rate_hash :rate_hash
      def rate_hash; site_rate_hash; end
    end
  end

  def site_rate_hash
    # compute new rate_hash
  end
end
```

In this example, the core rate\_hash method is aliased for later use. And the rate\_hash method is redefined inside the class\_eval block. This example demonstrates how to override the core shipping rate computation during checkout.

Next, I'll provide an example of extending an existing Spree model. site\_extension.rb will include the following:

```
...
def activate
  Product.send(:include, Spree::Site::Product)
end
...
```

And Spree::Site::Product module contains:

```
module Spree::Site::Product
  def new_product_method
    # new product instance method
  end
end
```

In the situation where you may want to create a class object method rather than an instance object method, you may include the following in Spree::Site::Product:

```
module Spree::Site::Product
  def self.included(target)
    def target.do_something special
```

```

    'Something Special!'
  end
end
end

```

The above example adds a method to the Product class object to be called from a view, for example `<%= Product.do_something_special %>` will return 'Something Special!!'.

To override a method from an existing model, I start with a module include in `site_extension.rb`:

```

...
def activate
  Product.send(:include, Spree::Site::Product)
end
...

```

And `Spree::Site::Product` contains the following:

```

module Spree::Site::Product
  def self.included(model)
    model.class_eval do
      alias :spree_master_price :master_price
      def master_price; site_master_price; end
    end
  end
  def site_master_price
    '1 billion dollars'
  end
end

```

And from the view, the two methods can be called within the following block:

```

<% @products.each do |product| -%>
  <%= product.master_price %> vs product.spree_master_price.to_s %>
<% end -%>

```

I previously discussed the introduction of hooks in depth [here](#) and [here](#). To extend an existing view that has a hook wrapped around the content you intend to modify, you may add something similar to the following to `*_hooks.rb`, where `*` is the extension name:

```

insert_after :homepage_products, 'shared/promo'

```

The above code inserts the 'shared/promo' view to be rendered above the `homepage_products` hook in the Spree gem or Spree source `~/app/views/products.index.html.erb` view. Other hook actions include `insert_before`, `replace`, or `remove`.

Before the introduction of hooks, the standard method of overriding or extending core views was to copy the core view into your extension view directory, and apply changes. In some cases, hooks are not always in the desired location. To override the footer view since there is no footer hook, I copy the Spree gem footer view to the extension view directory. The diff below compares the Spree gem view and my extension footer view:

```

- <div id="footer">
-   <div class="left">
-     <p>
-       <%= t("powered_by") %> <a href="http://spreecommerce.com/">Spree</a>
-     </p>
-   </div>
-   <div class="right">
-     <%= render 'shared/language_bar' if Spree::Config[:allow_locale_switching] %>
-   </div>
- </div>
<%= render 'shared/google_analytics' %>
+<p><a href="http://www.endpoint.com/">End Point</a></p>

```

## Sample data

A final tip that I've found helpful when developing with Spree is to create sample data files in the extension `db` directory to maintain data consistency between developers. In a recent project, I've created the following `stores.yml` data to initiate several stores for the multi domain extension:

```

store_1:
  name: Store1
  code: store1
  domains: store1.mysite.com
  default: false
store_2:
  name: Store2
  code: store2
  domains: store2.mysite.com
  default: true
store_3:
  name: Store3
  code: store3
  domains: store3.mysite.com
  default: false

```

Many of these tips apply to general to software development. The tips specific to development in Spree (and possibly other Rails platforms) include the sample data syntax and the described Ruby techniques to extend and override class and model functionality.

Learn more about End Point's [Ruby on Rails Development](#) or [Ruby on Rails Ecommerce Services](#).

## Spree on Heroku for Development

Yesterday, I worked through some issues to setup and run Spree on Heroku. One of End Point's clients is using Spree for a multi-store solution. We are using the **recently released Spree 0.10.0.beta gem**, which includes some significant Spree template and hook changes discussed [here](#) and [here](#) in addition to other substantial updates and fixes. Our client will be using Heroku for their production server, but our first goal was to work through deployment issues to use Heroku for development.



Since Heroku includes a free offering to be used for development, it's a great option for a quick and dirty setup to run

Spree non-locally. I experienced several problems and summarized them below.

## Application Changes

1. After a failed attempt to setup the basic Heroku installation described [here](#) because of a RubyGems 1.3.6 requirement, I discovered the need for [Heroku's bamboo deployment stack](#), which requires you to declare the gems required for your application. I also found the [Spree Heroku extension](#) and reviewed the code, but I wanted to take a more simple approach initially since the extension offers several features that I didn't need. After some testing, I created a .gems file in the main application directory including the contents below to specify the gems required on the Badious Bamboo Heroku stack.

```
rails -v 2.3.5
highline -v '1.5.1'
authlogic -v '>=2.1.2'
authlogic-oid -v '1.0.4'
activemerchant -v '1.5.1'
activerecord-tableless -v '0.1.0'
less -v '1.2.20'
stringex -v '1.0.3'
chronic -v '0.2.3'
whenever -v '0.3.7'
searchlogic -v '2.3.5'
will_paginate -v '2.3.11'
faker -v '0.3.1'
paperclip -v '>=2.3.1.1'
state_machine -v '0.8.0'
```

2. The next block I hit was that git submodules are not supported by Heroku, mentioned [here](#). I replaced the git submodules in our application with the Spree extension source code.

3. I also worked through addressing [Heroku's read-only filesystem](#) limitation. The setting perform\_caching is set to true for a production environment by default in an application running from the Spree gem. In order to run the application for development purposes, perform\_caching was set to false in RAILS\_APP/config/environments/production.rb:

```
config.action_controller.perform_caching = false
```

Another issue that came up due to the read-only filesystem constraint was that the Spree extensions were attempting to copy files over to the rails application public directory during the application restart, causing the application to die. To address this issue, I removed the public images and stylesheets from the extension directories and verified that these assets were included in the main application public directory.

I also removed the frozen Spree gem extension public files (javascripts, stylesheets and images) to prevent these files from being copied over during application restart. These files were moved to the main application public directory.

4. Finally, I disabled the allow\_ssl\_in\_production to turn SSL off in my development application. This change was made in the extension directory extension\_name\_extensions.rb file.

```
AppConfiguration.class_eval do
  preference :allow_ssl_in_production, :boolean, :default => false
end
```

Obviously, this isn't the preference setting that will be used for the production application, but it works for a quick and dirty Heroku development app. Heroku's SSL options are described [here](#).

## Deployment Tips

1. To create a Heroku application running on the Bamboo stack, I ran:

```
heroku create --stack bamboo-ree-1.8.7 --remote bamboo
```

2. Since my git repository is hosted on github, I ran the following to push the existing repository to my heroku app:

```
git push bamboo master
```

3. To run the Spree database bootstrap (or database reload), I ran the following:

```
heroku rake db:bootstrap AUTO_ACCEPT=1
```

As a side note, I ran the command heroku logs several times to review the latest application logs throughout troubleshooting.

Despite the issues noted above, the troubleshooting yielded an application that can be used for development. I also learned more about Heroku configurations that will need to be addressed when moving the project to production, such as SSL setup and multi domain configuration. We'll also need to determine the best option for serving static content, such as using Amazon's S3, which is supported by the Spree Heroku extension mentioned above.

Learn more about End Point's [Ruby on Rails Development](#) or [Ruby on Rails Ecommerce Services](#).

## Rails Ecommerce with Spree: Customizing with Hooks Comments

Yesterday, I went through some examples using [hook and theme implementation in Spree](#), an open source Ruby on Rails ecommerce platform. I decided to follow-up with closing thoughts and comments today.

I only spent a few hours working with the new Spree edge code (Version 0.9.5), but I was relatively happy with the Spree theme and hook implementation, as it does a better job decoupling the extension views with Spree core functionality and views. However, I found several issues that are potential areas for improvement with this release or releases to come.

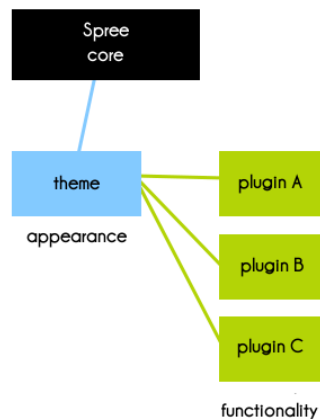
### Theme too clunky?

One concern I have is that the entire "views" directory from SPREE\_ROOT/app was moved into the theme with this theme-hook work (all of the "V" in MVC). Yesterday, I discussed how WordPress had designed a successful theme and plugin interaction and one thing I mentioned was that a WordPress theme was lightweight and comprised of several customer-facing PHP files (index, single post page, archive pages, search result page). Moving **all** of the Spree core views to the theme presents a couple of issues, in my opinion:

- A developer that jumps into theme development is immediately met with more than 50 files in the theme directory to understand and work with. What you may notice from my tutorial yesterday is that I actually changed the look of Spree through an extension rather than creating a new theme - I believe there is better separation of my custom design and the Spree core if I included the custom styling in the extension rather than creating a new theme and copying over 50+ files to edit. I'm also more comfortable working with CSS to manipulate the appearance rather than editing and maintaining those files. Now, the next time the Spree core and default template are updated, I don't have to worry about copying and pasting all the theme files into my custom theme and managing modifications. I think over time, Spree should aim to improve separation of theme views and core views and simplify the theme views.
- The new default Spree includes the admin views. Spree developers and users are probably more interested in changing and modifying customer-facing pages than admin pages. I believe that Spree should focus on developing a strong admin interface and assume that only more advanced developers will need to override the admin views. The admin view would contain a set of predefined core hooks to add tabs and reports. Rather than having a theme with all of the rails views, the theme should be a lightweight collection of files that are likely to be edited by users and the

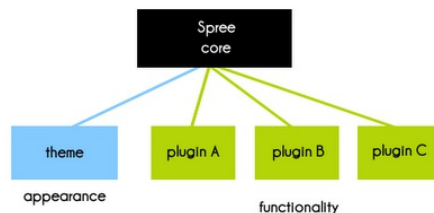
Spree core should include files that are less likely to be modified (and in theory, have an awesome admin interface that would only be extended with additional reports or additional fields for object updates and edits).

#### Theme-Hook Decoupling?



Extension views or text are hooked through the hooks defined in the theme.

Another big concern I have is the tight coupling between Spree themes and hooks. **All** of the hooks are defined in the Spree theme. If someone were to switch from one theme to another, there is the potential for functionality to be lost if consistency between theme hooks isn't enforced. This issue piggybacks off of the first issue: I think the Spree core should have control of all the admin views and admin hooks. It would be great to see the views simplified or refactored and allow Spree core to control and instantiate many hooks. I think it's great to provide the flexibility to instantiate hooks in themes, but I think the core code (admin, especially) should be more **opinionated** and contain its own set of views with hooks that would likely be overridden less frequently.



A more ideal approach to decouple appearance and functionality would require hooks to be defined in the Spree core.

#### Conclusion

In the tutorial, I also didn't address extended core functionality with models and controllers in the extensions. The logic discussed the article **Rails Ecommerce Product Optioning in Spree** and **Rails Approach for Spree Shopping Cart Customization** should work with some view modifications to use existing hooks instead of overriding core views.



A screenshot of the tutorial app in yesterday's article.

Despite the issues mentioned above, I think that the hook and theme work in the upcoming Spree 0.9.5 release is a big step in the right direction to improve the customization building blocks of Spree. It was mentioned in yesterday's article that the release hasn't been made official, but several developers have expressed an interest in working with the code. Hopefully the final kinks of theme and hook implementation will be worked out and the new release will be announced soon. Over time, the hook and theme implementation will advance and more examples and documentation will become available.

Learn more about End Point's general **Rails development** and **Rails shopping cart development**.

## Rails Ecommerce with Spree: Customizing with Hooks Tutorial

In the last couple months, there's been a bit of buzz around theme and hook implementation in **Spree**. The Spree team hasn't officially announced the newest version 0.9.5, but the edge code is available at <http://github.com/railsdog/spree> and developers have been encouraged to work with the edge code to check out the new features. Additionally, there is decent documentation [here](#) about theme and hook implementation. In this article, I'll go through several examples of how I would approach site customization using hooks in the upcoming Spree 0.9.5 release.

#### Background

I've been a **big proponent** of how WordPress implements themes, plugins, and hooks in the **spree-user Google group**. The idea behind WordPress themes is that a theme includes a set of PHP files that contain the display logic, HTML, and CSS for the customer-facing pages:

- index
- a post page
- archive pages (monthly, category, tag archives)
- search result page
- etc.

In many cases, themes include sections (referred to as partial views in Rails), or components that are included in multiple template pages. An example of this partial view is the sidebar that is likely to be included in all of the page types mentioned above. The WordPress theme community is abundant; there are many free or at-cost themes



available.

The concept behind WordPress plugins is much like Spree extension functionality - a plugin includes modular functionality to add to your site that is decoupled from the core functionality. Judging from the popularity of the WordPress plugin community, WordPress has done a great job designing the **Plugin API**. In most cases, the Plugin API is used to extend or override core functionality and add to the views without having to update the theme files themselves. An example of using the WordPress plugin API to add an action to the `wp_footer` hook is accomplished with the following code:

```
/* inside plugin */
function add_footer_text() {
    echo '<p>Extra Footer Text!!</p>';
}
add_action('wp_footer', 'add_footer_text');
```

WordPress themes and plugins with hooks are the building blocks of WordPress: with them, you piece together the appearance and functionality for your site. I reference WordPress as a basis of comparison for Spree, because like WordPress users, Spree users aim to piece together the appearance and functionality for their site. One thing to note is that the hook implementation in Spree is based on hook implementation in **Redmine**.

#### Spree Code

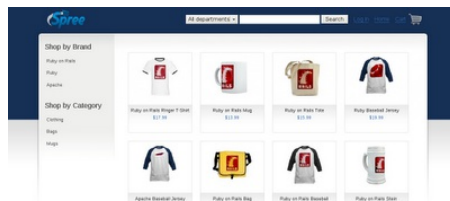
I grabbed the latest code at <http://github.com/railsdog/spree>. After examining the code and reviewing the SpreeGuides documentation, the first thing I learned is that there are four ways to work with hooks:

- insert before a hook component
- insert after hook component
- replace a hook component's contents
- remove a hook component

The next thing I researched was the hook components or elements. Below are the specific locations of hooks. The specific locations are more meaningful if you are familiar with the Spree views. The hooks are merely wrapped around parts of pages (or layouts) like the product page, product search, homepage, etc. Any of the methods listed above can act on any of the components listed below.

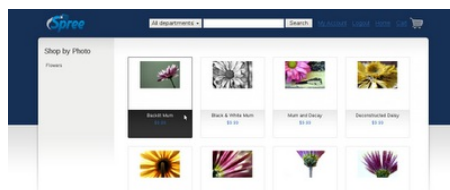
- layout: `inside_head`, `sidebar`
- homepage: `homepage_sidebar_navigation`, `homepage_products`
- product search: `search_results`
- taxon: `taxon_side_navigation`, `taxon_products`, `taxon_children`
- view product: `product_description`, `product_properties`, `product_taxons`, `product_price`, `product_cart_form`, `inside_product_cart_form`
- etc.

After I spent time figuring out the hook methods and components, I was ready to **do stuff**. First, I got Spree up and running (refer to the **SpreeGuides** for more information):



Spree startup with seed data and images.

Next, I updated the product list with a few pretend products. Let's take a quick look at the site with the updated products:



Spree with new product data for test site.

#### Example #1: Replace the logo and background styling.

First, I created an extension with the following code. Spree's extensions are roughly based off of **Radiant's** extension system. It's relatively simple to get an extension up and running with the following code (and server restart).

```
script/generate extension StepsPhotos
```

Next, I wanted to try out the `insert_after` method to append a stylesheet to the default theme inside the `<head>` html element. I also wanted to remove the sidebar because my test site only has 8 products (lame!) and I don't need sidebar navigation. This was accomplished with the following changes:

- First, I added the `insert_after` hook to add a view that contains my extra stylesheet. I also added the `remove` hook to remove the sidebar element:

```
# RAILS_ROOT/vendor/extensions/steps_photos/steps_photos_hooks.rb
insert_after :inside_head, 'shared/styles'
remove :sidebar
```

- Next, I added a new view in the extension to include the new stylesheet.

```
# RAILS_ROOT/vendor/extensions/steps_photos/app/views/shared/_styles.erb
<link type="text/css" rel="stylesheet" href="/stylesheets/steps_photos.css">
```

- Next, I created a new stylesheet in the extension.

```
/* RAILS_ROOT/vendor/extensions/steps_photos/public/stylesheets/steps_photos.css */
body { background: #000; }
body.two-col div#wrapper { background: none; }
a, #header a { color: #FFF; text-decoration: none; }

ul#nav-bar { width: 280px; line-height: 30px; margin-top: 87px; font-size: 1.0em; }
ul#nav-bar li form { display: none; }

.container { width: 750px; }
```

```
#wrapper { padding-top: 0px; }

.product-listing li { background: #FFF; height: 140px; }
.product-listing li a.info { background: #FFF; }

body#product-details div#wrapper { background: #000; }
body#product-details div#content, body#product-details div#content h1 { color: #FFF; margin-left: 10px; }
#taxon-crums { display: none; }
#product-description { width: 190px; border: none; }
.price.selling { color: #FFF; }
#product-image #main-image { min-height: 170px; }

/* Styling in this extension only applies to product and main page */

div#footer { display: none; }
```

One more small change was required to update the logo via a Rails preference. I set the logo preference variable to a new logo image and uploaded the logo to RAILS\_ROOT/vendor/extensions/stephs\_photos/public/images/.

```
# RAILS_ROOT/vendor/extensions/stephs_photos/stephs_photos_extension.rb
def activate
  AppConfiguration.class_eval do
    preference :logo, :string, :default => 'stephs_photos.png'
  end
end
```

After restarting the server, I was happy with the new look for my site accomplished using the `insert_after` and `remove` methods:



New look for Spree accomplished with several small changes.

*Note: You can also add a stylesheet with the code shown below. However, I wanted to use the hook method described above for this tutorial.*

```
def activate
  AppConfiguration.class_eval do
    preference :stylesheets, :string, :default => 'styles'
  end
end
```

#### Example #2: Use `insert_before` to insert a view containing Spree core functionality.

The next requirement I imagined was adding promo functionality to the product listing page. I wanted to use core Spree logic to determine which promo image to use. The first promo image would be a 10% off discount to users that were logged in. The second promo image would be a 15% off discount offered to users who weren't logged in and created an account. I completed the following changes for this work:

- First, I added the `insert_before` method to add the promo view before the `homepage_products` component, the component that lists the products on the homepage.

```
# RAILS_ROOT/vendor/extensions/stephs_photos/stephs_photos_hooks.rb
insert_before :homepage_products, 'shared/stephs_promo'
```

- Next, I added the view using core Spree user functionality.

```
# RAILS_ROOT/vendor/extensions/stephs_photos/app/views/shared/_stephs_promo.erb
<% if current_user -%>

<% else -%>

<% end -%>
```

- Finally, I uploaded my promo images to RAILS\_ROOT/vendor/extensions/stephs\_photos/public/images/

After another server restart and homepage refresh, I tested the logged in and logged out promo logic.



vs.



Spree core functionality used to display two different promo images inside a partial view.

*Note: The promo coupon logic that computes the 10% or 15% off was not included in this tutorial.*

#### Example #3: Use `replace` method to replace a component on all product pages.

In my third example, I imagined that I wouldn't have time to manage product descriptions when I was rich and famous. I decided to use the `replace` hook to replace the product description on all product pages. I completed the

following steps for this change:

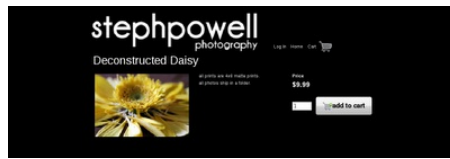
- First, I added the `replace` method to replace the `:product_description` component with a rails partial view.

```
# RAILS_ROOT/vendor/extensions/stephs_photos/stephs_photos_hooks.rb
replace :product_description, 'shared/generic_product_description'
```

- Next, I created the view with the generic product description.

```
# RAILS_ROOT/vendor/extensions/stephs_photos/app/views/shared/_generic_product_description.erb
all prints are 4x6 matte prints.<br />
all photos ship in a folder.
```

After yet another server restart and product refresh, I tested the generic product description using the replace hook.



The `replace` hook was used to replace product descriptions on all product pages.

#### Intermission

OK, so hopefully you see the trend:

1. Figure out which component you want to pre-append, post-append, replace, or remove.
2. Modify `extension_name_hooks.rb` to include your hook method (and pass the view, if necessary).
3. Create the new view in your extension.
4. Restart server and be happy!

I'll note a couple other examples below.

#### Example #4: Bummer that there's no footer component

In the next step, I intended to add copyright information to the site's footer. I was disappointed to find that there was no hook wrapped around the footer component. So, I decided not to care for now. But in the future, my client (me) may make this a higher priority and the options for making this change might include:

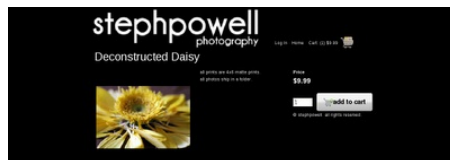
- Clone the default template and modify the template footer partial view.
- Clone the default template, create a hook to wrap around the footer component, add the changes via a hook in an extension.
- Add a view in the extension that overrides the theme footer view.

#### Example #5: Add text instead of partial view.

Since I couldn't add copyright information below the footer, I decided to add it using `after` the `inside_product_cart_form` component using the `insert_after` hook. But since it's a Friday at 5:30pm, I'm too lazy to create a view, so instead I'll just add text for now with the following addition to the extension hooks file:

```
# RAILS_ROOT/vendor/extensions/stephs_photos/stephs_photos_hooks.rb
insert_after :inside_product_cart_form, :text => '<p>&copy; stephpowell. all rights reserved.</p>'
```

Server restart, and I'm happy, again:



Text, rather than a partial view, was appended via a hook.

Hopefully my examples were exciting enough for you. There's quite a lot you can do with the hook methods, and over time more documentation and examples will become available through the Spree site, but I wanted to present a few very simple examples of my approach to customization in Spree. I've uploaded the extension to [http://github.com/stephskardal/stephs\\_photos](http://github.com/stephskardal/stephs_photos) for this article.


Tomorrow, I'm set to publish closing thoughts and comments on the hook implementation since this article is now too long for a blog post. Stay tuned.

Learn more about End Point's general **Rails development** and **Rails shopping cart development**.

## Rails Ecommerce Product Optioning in Spree

A couple of months ago, I worked on a project for **Survival International** that required two-dimensional product optioning for products. The shopping component of the site used **Spree**, an open source rails ecommerce project that End Point previously sponsored and continues to support. Because the Spree project is quickly evolving, we wanted to implement a custom solution that would "stand the test of time" and work with new releases. I worked with the existing data structures and functionality as much as possible. The product optioning implementation discussed in this article should translate to other ecommerce platforms as well.

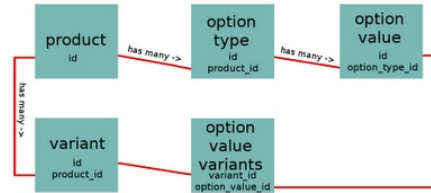
Choose your colour, size and quantity below.

	S	M	L	XL
Heather grey and black	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Olive and black	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Quantity	<input type="text" value="1"/>			
 Add to shopping basket				

Here's what I mean when I say "two dimensional product optioning".

The first step to extending the core ecommerce functionality was to understand the data model. A single product "has many" option types (size, color). An option type "has many" option values (size: small, medium, large). Each product also "has many" variants. Each variant was tied to an option value for each product option type. For example, each variant would require a corresponding size and color option value in the example above. Ideally, each variant

represents a unique size and color combination.



An \*awesome\* database dependency diagram.

Using the Spree demo data, I set up the Apache Baseball Jersey to have option types "PO\_Size" and "PO\_Color". PO\_Size contains option values Red, Blue, and Green. PO\_Color contains option values Small, Medium, and Large.

#### Editing Product "Apache Baseball Jersey"

Options	Price	SKU	Weight	Height	Width	Depth	On Hand	Action
PO_Color Red, PO_Size S	19.99	smallred				10		<a href="#">Edit</a> <a href="#">Delete</a>
PO_Color Red, PO_Size M	19.99	mediumred				10		<a href="#">Edit</a> <a href="#">Delete</a>
PO_Color Blue, PO_Size S	19.99	smallblue				0		<a href="#">Edit</a> <a href="#">Delete</a>
PO_Color Green, PO_Size S	19.99	smallgreen				10		<a href="#">Edit</a> <a href="#">Delete</a>
PO_Color Blue, PO_Size M	19.99	mediumblue				10		<a href="#">Edit</a> <a href="#">Delete</a>
PO_Color Green, PO_Size M	19.99	mediumgreen				10		<a href="#">Edit</a> <a href="#">Delete</a>
PO_Color Red, PO_Size L	19.99	largered				10		<a href="#">Edit</a> <a href="#">Delete</a>
PO_Color Blue, PO_Size L	19.99	largeblue				10		<a href="#">Edit</a> <a href="#">Delete</a>
PO_Color Green, PO_Size L	19.99	largegreen				0		<a href="#">Edit</a> <a href="#">Delete</a>

Variants assigned to the Apache Baseball Jersey

The second step to producing a two dimensional product option table was to generate the required data in a before\_filter method in the controller. Below are the contents of the module that generates the hash in the before\_filter method with color and size information. The module retrieves active variants first, then verifies that the required option types are tied to the product. Then, size, color, and variant ids are collected from the active variants producing the data structure described above.

```
def self.included(target)
  target.class_eval do
    before_filter :define_2d_option_matrix, :only => :show
  end
end

def define_2d_option_matrix
  variants = Spree::Config[:show_zero_stock_products] ?
    object.variants.active.select { |a| !a.option_values.empty? } :
    object.variants.active.select { |a| !a.option_values.empty? && a.in_stock }
  return if variants.empty? ||
    object.option_types.select { |a| a.presentation == 'PO_Size' }.empty? ||
    object.option_types.select { |a| a.presentation == 'PO_Color' }.empty?
  variant_ids = Hash.new
  sizes = []
  colors = []
  variants.each do |variant|
    active_size = variant.option_values.select { |a| a.option_type.presentation == 'PO_Size' }.first
    active_color = variant.option_values.select { |a| a.option_type.presentation == 'PO_Color' }.first
    variant_ids[active_size.id.to_s + '_' + active_color.id.to_s] = variant.id
    sizes << active_size
    colors << active_color
  end
  size_sort = Hash['S', 0, 'M', 1, 'L', 2]
  @sc_matrix = { 'sizes' => sizes.sort_by { |s| size_sort[s.presentation] }.uniq,
    'colors' => colors.uniq,
    'variant_ids' => variant_ids }
end
```

The code above produces a hash with three components:

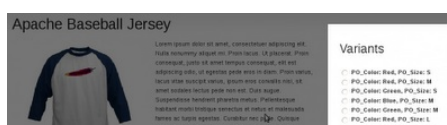
- @sc\_matrix['variant\_ids']: a hash that maps size and color combinations to variant id
- @sc\_matrix['sizes']: an array of sorted unique sizes of product variants
- @sc\_matrix['colors']: an array of unique colors of product variants

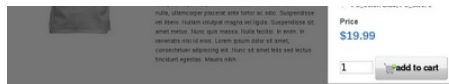
In the view, the output of size and color arrays is used to generate a table. In this hardcoded view, sizes are displayed as the horizontal option across the top of the table, and colors as the vertical option along the left side of the table.

```
...
<% if @sc_matrix -%>
  <p>Choose your colour, size and quantity below.</p>
  <table id="option-matrix">
    <tr>
      <th></th>
      <% @sc_matrix['sizes'].each do |s| %>
        <th class="size"><%= s.presentation %></th>
        <td class="spacer"></td>
      <% end -%>
    </tr>
    <% @sc_matrix['colors'].each do |c| -%>
      <tr>
        <th class="color"><%= c.presentation %></th>
        <% @sc_matrix['sizes'].each do |s| -%>
          <td>
            <% if @sc_matrix['variant_ids'][s.id.to_s + '_' + c.id.to_s] -%>
              <input type="radio" value="<%= @sc_matrix['variant_ids'][s.id.to_s + '_' + c.id.to_s] %>" name="products[<%= @product.id %>]" />
            <% else -%>
              
            <% end -%>
          </td>
        <td class="spacer"></td>
      <% end -%>
    </tr>
  </table>
<% elsif #check for other stuff
...

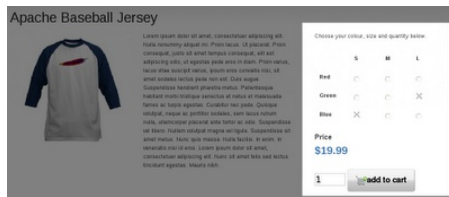
```

Here is a comparison of the current variant display method versus two dimensional variant display of the same product:





Current variant display method.



Two dimensional variant display method. Two variants shown here are out of stock.

And here is another example of two dimensional optioning in use at Survival International (more glamorous styling):

Choose your colour, size and quantity below.

M L XL  
 Dijon and chocolate ☒ ☐ ☐  
 Silver and graphite ☐ ☐ ☐  
 Quantity

Spree extensions are similar to WordPress plugins or Drupal modules that do not typically require you to edit core code. The primary components of the extension are a module with the `before_filter` functionality and a custom view that overrides the core product view. An extension was created for this functionality and it lives at <http://github.com/stephskardal/spree-product-options>.

Possibilities for future work include editing the extension to be more robust by eliminating the use of the hard-coded option types of "PO\_Size" and "PO\_Color" and removing the hard-coded size ordering hash. It would be ideal to be able to assign the two dimension option types (horizontal axis and vertical axis) in the Spree admin for each product or a set of products. Another option for future work with this extension includes extending the functionality to multi-dimensional product optioning that would allow you to select more than two option types per product (for example: size, color, and material), but this functionality is more complex and may be dependent on JavaScript to hide and show option types and values.

Learn more about End Point's general [Rails development](#) and [Rails shopping cart development](#).

## Rails Approach for Spree Shopping Cart Customization

Recently, I was assigned a project to develop **Survival International's** ecommerce component using **Spree**. **Survival International** is a non-profit organization that supports tribal groups worldwide in education, advocacy and campaigns. Spree is an open source Ruby on Rails ecommerce platform that was sponsored by End Point from its creation in early 2008 until May 2009, and that we continue to support. End Point also offers a **hosting solution for Spree (SpreeCamps)**, that was used for this project.

Spree contains ecommerce essentials and is intended to be extended by developers. The project required customization including significant cart customization such as adding a buy 4 get 1 free promo discount, adding free giftwrap to the order if the order total exceeded a specific preset amount, adding a 10% discount, and adding a donation to the order. Some code snippets and examples of the cart customization in rails are shown below.

An important design decision that came up was how to store the four potential cart customizations (buy 4 get 1 free promo, free giftwrap, 10% discount, and donation). The first two items (4 get 1 free and free gift wrap) are dependent on the cart contents, while the latter two items (10% discount and donation) are dependent on user input. Early on in the project, I tried using session variables to track the 10% discount application and donation amount, and I applied an `after_filter` to calculate the buy 4 get 1 free promo and free giftwrap for every order edit, update, or creation. However, this proved somewhat cumbersome and required that most Rails views be edited (frontend and backend) to show the correct cart contents. After discussing the requirements with a coworker, we came up with the idea of using a single product with four variants to track each of the customization components.

I created a migration file to introduce the following variants similar to the code shown below. A single product by the name of 'Special Product' contained four variants with SKUs to denote which customization component they belonged to ('supporter', 'donation', 'giftwrap', or '5cards').

```
p = Product.create(:name => 'Special Product', :description => "Discounts, Donations, Promotions", :master_price => 1.00)
v = Variant.create(:product => p, :price => 1.00, :sku => 'supporter') # 10% discount
v = Variant.create(:product => p, :price => 1.00, :sku => 'donation') # donation
v = Variant.create(:product => p, :price => 1.00, :sku => 'giftwrap') # free giftwrap
v = Variant.create(:product => p, :price => 1.00, :sku => '5cards') # buy 4 get 1 free discount
```

Next, I added accessor elements to retrieve the variants shown below. Each of these accessor methods would be used throughout the code and so this would be the only location requiring an update if the variant SKU was modified.

```
module VariantExtend
  ...
  def get_supporter_variant
    Variant.find_by_sku('supporter')
  end
  def get_donation_variant
    Variant.find_by_sku('donation')
  end
  def get_giftwrap_variant
    Variant.find_by_sku('giftwrap')
  end
  def get_cards_promo_variant
    Variant.find_by_sku('5cards')
  end
  ...
end
```

The design to use variants makes the display of cart contents on the backend and frontend much easier, in addition to calculating cart totals. In Spree, the line item price is not necessarily equal to the variant price or product master price, so the prices stored in the product and variant objects introduced above are meaningless to individual orders. An `after_filter` was added to the Spree orders controller to add, remove, or recalculate the price for each special product variant. The order of the `after_filters` was important. The cards (buy 4 get 1 free) discount was added first, followed by a subtotal check for adding free giftwrap, followed by adding the supporter discount which reduces the total price by 10%, and finally a donation would be added on top of the order total:

```

OrdersController.class_eval do
  after_filter [:set_cards_discount, :set_free_giftwrap, :set_supporter_discount, :set_donation], :only => [:create, :edit, :update]
end

```

Each after filter contained specific business logic. The cards discount logic adds or removes the variant from the cart and adjusts the line item price:

```

def set_cards_discount
  v = Variant.new.get_cards_promo_variant # get variant
  # calculate buy 4 get 1 free discount (cards discount)
  # remove variant if order contains variant and cards_discount is 0
  # add variant if order does not contain variant and Cards_discount is not 0
  # adjust price of discount line item to cards_discount
  # save order
end

```

The free giftwrap logic adds or removes the variant from the cart and sets the price equal to 0:

```

def set_free_giftwrap
  v = Variant.new.get_giftwrap_variant # get variant
  # remove variant if cart contains variant and order subtotal < 40
  # add variant if cart does not contain variant and order subtotal >= 40
  # adjust price of giftwrap line item to 0.00
  # save order
end

```

The supporter discount logic adds or removes the discount variant depending on user input. Then, the line item price is adjusted to give a 10% discount if the cart contains the discount variant:

```

def set_supporter_discount
  v = Variant.new.get_supporter_variant # get variant
  # remove variant if cart contains variant and user input to receive discount is 'No'
  # add variant if cart does not contain variant and user input to receive discount is 'Yes'
  # adjust price of discount line item to equal 10% of the subtotal (minus existing donation)
  # save order
end

```

Finally, the donation logic adds or removes the donation variant depending on user input:

```

def set_donation
  v = Variant.new.get_donation_variant # get variant
  # remove variant if cart contains variant and user donation is 0
  # add variant if cart does not contain variant and user donation is not 0
  # adjust price of donation line item
  # save order
end

```

This logic results in a simple process for all four variants to be adjusted for every recalculation or creation of the cart. Also, the code examples above used existing Spree methods where applicable (add\_variant) and created a few new methods that were used throughout the examples above (Order.remove\_variant(variant), Order.adjust\_price(variant, price)). A few changes were made to the frontend cart view.

The screenshot shows a web interface for a store named 'Survival'. At the top, there are navigation links: home, information, contact, catalogue, and Survival. Below this is a table of line items with columns: Item, Qty, Price, Quantity, and Amount. The first item is 'Reindeer in book' with a quantity of 3 and a price of £14.85, totaling £44.55. The second item is 'Ice cream' with a quantity of 2 and a price of £4.95, totaling £9.90. Below the line items, there are sections for 'giftwrap' and 'Card Discount'. The 'Sub-total' is £59.40. There is a section for 'Apply supporter discount?' with a dropdown menu set to 'Yes'. Below this, it says 'Your supporter discount is £-5.94' and 'Find out more about your supporter discount'. There is also a section for 'Make a donation' with a text input field set to '10.00'. The final 'Total' is £63.46. A note at the bottom left states 'All prices are in UK pounds.'.

To render the desired view, line items belonging to the "Special Product" were not displayed in the default order line display. The buy 4 get 1 free promo and free giftwrap were added below the default line order items. Donations and discounts were shown below the line items in order of how they are applied to the order. The backend views were not modified and as a result the site administrators would see all special variants in an order:

**Order R414156870** Refresh Cancel

Item Description	Price	Qty	Total
Card One	£4.85	3	£14.55
Card Two	£4.85	12	£58.40
Special Product (Special Product christmas card discount)	£14.85	1	£14.85
Special Product (Special Product free gift-wrap)	£0.00	1	£0.00
Special Product (Special Product supporter discount)	£-5.94	1	£-5.94
Special Product (Special Product donation)	£10.00	1	£10.00
<b>Subtotal</b>			£59.40
<b>Tax:</b>			£0.00
<b>Shipping (Only one option)</b>			£12.06
<b>Order Total:</b>			£76.45

**Bill Address**  
 Steph Powell  
 123 Main Ave  
 Rugeley, UK B81 1H  
 United Kingdom  
 123-123-123

**Ship Address**  
 Steph Powell  
 123 Main Ave  
 Rugeley, UK B81 1H  
 United Kingdom  
 123-123-123

An additional method was created to define the total number of line items in the order, shown at the top right of every page except for the cart and checkout page.

The screenshot shows the top of the 'Survival' website. It features a navigation bar with links: home, information, contact, catalogue, and Survival. To the right of the navigation bar is a search bar with a 'Search' button and a link to '13 Items (incl. tax)'. Below the navigation bar, there is a section for 'module OrderExtend'.

```

module OrderExtend
  ...
  def mod_num_items
    item_count = line_items.inject(0) { |kount, line_item| kount + line_item.quantity } +
      (contains?(Variant.new.get_supporter_variant) ? -1 : 0) +
      (contains?(Variant.new.get_donation_variant) ? -1 : 0) +
      (contains?(Variant.new.get_giftwrap_variant) ? -1 : 0) +
      (contains?(Variant.new.get_cards_promo_variant) ? -1 : 0)
    item_count.to_s + (item_count != 1 ? ' items' : ' item')
  end
  ...
end

```

The solution developed for this project was simple and extended the Spree core ecommerce code elegantly. The complex business logic required was easily integrated in the variant accessor methods and after\_filters to re-add, remove, and recalculate the price of the custom variants where necessary. The project required additional

customizations, such as view modifications, navigation modifications, and complex product optioning, which may be discussed in future blog posts :).

Learn more about End Point's **rails development** or **rails ecommerce development**.

## Packaging Ruby Enterprise Edition into RPM

It's unfortunate that past versions of Ruby have gained a reputation of **performing poorly**, consuming too much memory, or otherwise being "**unfit for the enterprise**." According to the fine folks at Phusion, this is partly **due to the way Ruby does memory management**. And they've created an alternative branch of Ruby 1.8 called "**Ruby Enterprise Edition**." This code base includes **many significant patches** to the stock Ruby code which dramatically improve performance.

Phusion **advertises an average memory savings** of 33% when combined with **Passenger**, their Apache module for serving Rails apps. We did some testing of our own, using virtualized Xen servers from our **Spreecamps.com** offering. These servers use the **DevCamps** system to run several separate instances of httpd for each developer, so reducing the usage of Passenger was crucial to fitting into less than a gigabyte of memory. Our findings were dramatic: one instance dropped 100MB down to 40MB. (The **status tools** included with Passenger were very helpful in confirming this.)

There has been some discussion on the **Phusion Passenger** and **other mailing lists** about packaging **Ruby Enterprise Edition** for Red Hat Enterprise Linux and its derivatives (**CentOS** and **Fedora**). **Packages are available** from Phusion for Ubuntu Linux, but many of our clients prefer RHEL's reputation as a stable platform for e-commerce hosting. So we've packaged ruby-enterprise into RPM and made them available to give back to the Rails community.

We want our SpreeCamps systems to be easy to maintain, following the "**Principle of Least Astonishment**." By default, Phusion's script installs ruby-enterprise into `/opt`, so invocation must include the full path to the executable. This would be unsettling to a developer who mistakenly installed gems to Red Hat's rubygems path while intending to install gems usable by REE and Passenger. It is important to install the `ruby` and `gem` executables into all users' `$PATH`.

We took a cue from our customized local-perl packages. These packages install themselves into `/usr/local`. This means that all executables reside in `/usr/local/bin`; no `$PATH` modifications are necessary to utilize them via the command-line. Our ruby-enterprise packages are configured the same way. (If another `/usr/local/bin/ruby` exists, package installation will fail before clobbering another ruby installation.) Applications which specify `#!/usr/bin/ruby` will continue to use Red Hat's packaged ruby.

Similar to a source-based installation, once these packages are installed you may do `gem install passenger` and any other gems your application needs. Phusion's REE installer also installs several "useful gems". However we elected not to include these in the main `ruby-enterprise` RPM package. More, smaller packages limited to a particular module or piece of software, is better than one or two big fat RPMs with a bunch of stuff you may or may not need. We will likely package individual gems in the near future.

These packages are publicly available from our repository. We've just begun using these but are finding them reliable and very helpful so far. Any of you who would like to are welcome to try them out via direct download, or much easier, adding our Yum repository to your system as described here:

<https://packages.endpoint.com/>

Once you've done that, a simple command should get you most of the way there:

```
yum install ruby-enterprise ruby-enterprise-rubygems
```

If you prefer to download them directly, the .rpm packages are available on that site as well, just browse through the repo.

The .spec file is available for review and forking on GitHub: <http://gist.github.com/108940>

Many thanks to list member Tim Charper for providing an example .spec, and my colleagues at End Point for reviewing this work.

We appreciate any comments or questions you may have. This package repo is for us and our clients primarily, but if there's a package you need that isn't in there, let us know and maybe we'll add it.

## Git rebase: Just-Workingness Baked Right In (If you're cool enough)

Reading about rebase makes it seem somewhat abstract and frightening, but it's really pretty intuitive when you use it a bit. In terms of how you deal with merging work and addressing conflicts, rebase and merge are very similar.

Given branch "foo" with a sequence of commits:

```
foo: D --> C --> B --> A
```

I can make a branch "bar" off of foo: (`git branch bar foo`)

```
foo: D --> C --> B --> A
bar: D --> C --> B --> A
```

Then I do some development on bar, and commit. Meanwhile, somebody else develops on foo, and commits. Introducing new, unrelated commit structures.

```
foo: E --> D --> C --> B --> A
bar: X --> D --> C --> B --> A
```

Now I want to take my "bar" work (in commit X) and put it back upstream in "foo".

- I can't push from local bar to upstream foo directly because it is not a fast-forward operation; foo has a commit (E) that bar does not.
- I therefore have to either merge local bar into local foo and then push local foo upstream, or rebase bar to foo and then push.

A merge will show up as a separate commit. Meaning, merging bar into foo will result in commit history:

```
foo: M --> X --> D --> C --> B --> A
    |
    +--> E --> D --> C --> B --> A
```

(The particulars may depend on conflicts in E versus X).

Whereas, from branch "bar", I could "`git rebase foo`". Rebase would look and see that "foo" and "bar" have commits in common starting from D. Therefore, the commits in "bar" more recent than D would be pulled out and applied on top of the full commit history of "foo". Meaning, you get the history:

```
bar: X' --> E --> D --> C --> B --> A
```

This can be pushed directly to "foo" upstream because it contains the full "foo" history and is therefore a fast-forward operation.

Why does X become X' after the rebase? Because it's based on the original commit X, but it's not the same commit; part of a commit's definition is its parent commit, and while X originally referred to commit D, this derivative X' refers

instead to E. The important thing to remember is that the content of the X' commit is taken initially from the original X commit. The "diff" you would see from this commit is the same as from X.

If there's a conflict such that E and X changed the same lines in some file, you would need to resolve it as part of rebasing, just like in a regular merge. But those changes for resolution would be part of X', instead of being part of some merge-specific commit.

## Considerations for choosing rebase versus merge

Rebasing should generally be the default choice when you're pulling from a remote into your repo.

```
git pull --rebase
```

Note that it's possible to make --rebase the default option for pulling for a given branch. From Git's pull docs:

To make this the default for branch *name*, set configuration `branch.name.rebase` to true.

However, as usual with Git, saying "do this by default" only gets you so far. If you assume rebase is always the right choice, you're going to mess something up.

Probably the most important rule for rebasing is: do not rebase a branch that has been pushed upstream, unless you are positive nobody else is using it.

Consider:

- **Steph** has a **Spree fork on Github**. So on her laptop, she has a repo that has her Github fork as its "origin" remote.
- She also wants to easily pull in changes from the canonical Spree Github repo, so she has that repo set up as the "canonical" remote in her local repo.
- Steph does work on a branch called "address\_book", unique to her Github fork (not in the canonical repo).
- She pushes her stuff up to "address\_book" in origin.
- She decides she needs the latest and greatest from canonical. So she fetches canonical. She can then either: rebase canonical/master into address\_book, or merge.

The merge makes for an ugly commit history.

The rebase, on the other hand, would make her local address\_book branch incompatible with the upstream one she pushed to in her Github repo. Because whatever commits she pushed to origin/address\_book that are specific to that branch (i.e. not on canonical/master) will get rebased on top of the latest from canonical/master, meaning they are now **different** commits with a different commit history. Pushing is now not really an option.

In this case, making a different branch would probably be the best choice.

Ultimately, the changes Steph accumulates in address\_book should indeed get rebased with the stuff in canonical/master, as the final step towards making a clean history that could get pulled seamlessly onto canonical/master.

So, in this workflow, a final step for publishing a set of changes intended for upstream consumption and potential merge into the main project would be, from Steph's local address\_book branch:

```
# get the latest from canonical repo
git fetch canonical
# rebase the address book branch onto canonical/master
git rebase canonical/master
# work through any conflicts that may come up, and naturally test
# your conflict fixes before completing
...
git push origin address_book:refs/heads/address_book_release_candidate
```

That would create a branch named "address\_book\_release\_candidate" on Steph's Github fork, that has been structured to have a nice commit history with canonical/master, meaning that the Spree corefolks could easily pull it into the canonical repo if it passes muster.

What you would not **ever** do is:

```
git fetch canonical
# make a branch based off of canonical/master
git branch canonical_master canonical/master
# rebase the master onto address_book
git rebase address_book
```

As that implies messing with the commit history of the canonical master branch, which we all know to be published and therefore must not be subject to history-twiddling.

## Rails Conf 2009 - Company Report

RailsConf 2009 concluded last week so its time for me to talk about some of the highlights for my fellow teammates that could not make it. I think one of the more interesting talks was given by **Yehuda Katz**. The talk was on the "Russian Doll Pattern" and dealt with mountable apps in the upcoming Rails 3.0 release (slides available [here](#).) Even though he felt like it wasn't his best talk I thought it was quite interesting. Personally, I thought it was refreshing to see something that was not yet complete. The Rails core team should do more of this kind of thing as it provides the community a chance to give feedback on features before they're set in stone.

The **Rails Envy** guys were there and gave a very interesting presentation about innovations in Rails this past year. I was pleasantly surprised to see that **Spree** made this list and they even included the new interface in their screenshots. Gregg made some excellent videos from the conference as well which capture some of the spirit of the conference.

This year I had a chance to meet Fabio Akita in person. Fabio has a great blog called **Akita on Rails** which has a huge following in Brazil. He also does a lot of interesting in-depth interviews with people. This year I had the honor of being the subject of one of those interviews. I basically talked about the origins of the Spree project and what makes it special. You can listen to this and his other interviews [here](#).

One of the more whimsical talks was about Ruby performance. The Phusion guys (who brought us Passenger) created a hilarious version of Wolfenstein written entirely in Ruby. Gregg Pollack put up a **fun video** showing how it looked. Supposedly this was to show off how Ruby can in fact scale but it seemed more like an excuse to make a lot of inside jokes. Even so, one or two fun presentations like this tend to liven up what would otherwise be a pretty dull conference.

The Spree BOF talk went well. It was very informal and we used it as a chance to meet some of our users and to find out what kinds of features they would like to see in future versions Spree. Steph already gave an excellent summary of this so I won't rehash it here.

Overall this was a great conference. Las Vegas was a pretty good location due to its cheap hotels and convenient flights. The hotel itself was not the best and the cigarette smoking was completely out of hand. I have it on good information that the conference will be in a different location next year. Rumor has it that it may even be on the east coast. Wherever it is, I definitely look forward to going back next year!

Learn more about End Point's [rails development](#) and [spree development](#).



## Spree at RailsConf

Last week at **RailsConf 2009**, the Spree folks from End Point conducted a **Birds of a Feather** session to discuss **Spree**, an End Point sponsored open source rails ecommerce platform. Below is some of the dialog from the discussion (paraphrased).

Crowd: "How difficult is it to get Spree up and running from start to finish?"

Spree Crew: "This depends on the level of customization. If a customer simple needs to reskin the site, this shouldn't take more than a week (hopefully much less than a full week). If the customer needs specific functionality that is not included in core functionality or extensions, you may need to spend some time developing an **extension**."

Crowd: "How difficult is it to develop extensions in Spree?"

Spree Crew: "Spree extension work is based on the work of the **Radiant** community. Extensions are mini-applications: they allow you to drop a pre-built application into spree to override or insert new functionality. Documentation for extensions is available at the **spree github wiki**. We also plan to release more extensive Spree Guides documentation based on **Rails Guides** soon."

Spree Crew: "How did you hear about Spree?"

Crowd: "My client and I found it via search engines. My client thought that Spree looked like a good choice."

Spree Crew: "What other platforms did you consider before you found spree?"

Crowd: "Magento", "Substruct", "My client considered Magento, but I know several people that have developed with Magento and have found it difficult to override core functionality."

Spree Crew: "What types of functionality were missing from Spree that you'd like to see developed in the future?"

Crowd: "My client wanted checkout split into multiple steps instead of the new single page checkout. I was able to implement this by overriding the Spree checkout library and checkout views.", "My client needed complex inventory management.", "My client needed split shipping functionality."

Crowd: "What is the plan for Spree with regards to CMS development?"

Spree Crew: "There has been some discussion on the **integration of a CMS into Spree**. No one in the Spree community appears to be currently working on this. Contributions in this area are welcome. Also, Yehuda Katz is giving a talk on **mountable apps** - the Spree community would like to investigate the implications this has for Spree."

Crowd: "What are the next steps for localization, especially multilingual product descriptions?"

Spree Crew: "This is on the radar for future Spree development. It is not currently in development, and again, contributions in this area are welcome."

From the discussion, I took away that some of the desired features for Spree are inventory management, split shipping functionality, cms integration, and improved localization. I hope that the application of Spree continues to contribute to it's progress. The Spree Crew also hopes to showcase some of the sites referenced above at the spree site.

Learn more about End Point's **rails development** and **rails shopping cart development**.

## Stuff you can do with the PageRank algorithm

I've attended several interesting talks so far on my first day of RailsConf, but the one that got me the most excited to go out and start trying to shoehorn it into my projects was **Building Mini Google in Ruby** by Ilya Grigorik.

In terms of doing Google-like stuff (which I'm not especially interested in doing), there are three steps, which occur in order of increasing level of interestingness. They are:

- Crawling (mundane)
- Indexing (sort of interesting)
- Rank (neato)

Passing over crawling, Indexing is sort of interesting. You can do it yourself if you care about the problem, or you can hand it over to something like ferret or sphinx. I expect it's probably time for me to invest some time investigating the use of one or more of these, since I've already gone up and down the do it yourself road.

The interesting bit, and the fascinating focus of Ilya's presentation were the explanation of the PageRank algorithm and the implementation details as well as some application ideas. Hopefully I don't mess this up too badly, but as I understand it, it simplifies down to something like this.

A page is ranked to some degree by how many other pages link to it. This is a bit too simple, though, and trivially gamed. So, you make it a little more complex by modeling the following behavior, a *random surfer* will surf from one page to another by doing one of two things. They will either follow a link or randomly go to a non-linked page (sort of how I surf Wikipedia). There is a much higher probability (.85) that they will follow a link than that they will teleport (.15). If you model this (hand waving here) then you come up with a nice formula (more hand waving) that can be used to calculate the page rank for a page in a given data set. A data set in this case is a collection of crawled pages.

For large data sets, these calculations can be somewhat intensive, so we are recommended to the good graces of the Gnu Scientific Library and the appropriate Ruby wrappers and the NArray gem to do the calculations and array management.

One suggestion of a practical applications of this technology is to apply it to sets of products purchased together in a shopping cart to provide recommendations of the sort for 'people who bought that also bought this.' I'm pretty excited to try to implement this in **Spree**. But...

...what really piqued my interest was the idea that this could be applied to any graph. The Taxonomies/Taxons/ProductGroups with products could give me a nice big (depending on the size of the data set of course) directed graph to play with. The question, I suppose, is what the PageRank applied against such a graph means.

## Announcing SpreeCamps.com hosting

On day two of **RailsConf 2009**, we are pleased to announce our new **SpreeCamps.com** hosting service. SpreeCamps is the quickest way to get started developing your new e-commerce website with Ruby on Rails and Spree and easily deploy it into production.

You get the latest Spree 0.8.0 that was just released yesterday, as part of a fully configured environment built on the best industry-standard open-source software: CentOS, Ruby on Rails, your choice of PostgreSQL or MySQL, Apache, Passenger, Git, and DevCamps. Your system is harmonized and pre-installed on high-performance hardware, so you can simply sign up and start coding today.

SpreeCamps gives you a 64-bit virtual private server and include backups, your own preconfigured iptables firewall, ping and ssh monitoring, and DNS. We also include a benefit unheard of in the virtual private server space: Out of the box we enforce an SELinux security policy that protects you against many types of unforeseen security vulnerabilities, and is configured to work with Passenger, Rails, and Spree.

SpreeCamps' built-in DevCamps system gives you development and staging environments that make it easy to work together in teams, show others your work in progress, and deploying your changes from development to production

is as easy as "git pull".

And the best part of all? You can sign up now for just \$95 per month, with no setup charge. We also offer hands-on training and support for Spree, DevCamps, and any other part of your application. Visit [SpreeCamps.com](http://SpreeCamps.com) for more information or to sign up for your own SpreeCamp.

(By the way, it works fine for hosting any other kind of Rails application, or whatever else you want too. We've already used it for one non-Rails website because it made getting going with DevCamps so easy.)

Learn more about End Point's [rails development](#) and [rails shopping cart development](#).

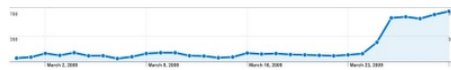
## Rails Conf Kicking Off in Less Than 24 Hours

Less than 24 hours until RailsConf kicks off. End Point is going to be there in force this year with three of its Spree contributors attending (myself included.) Looking forward to seeing everyone out there. We'll also be making a few big Spree related announcements so stay tuned! If you're a twitter user, follow me [@railsdog](#). Follow Spree [@spreecommerce](#). I'll also be blogging the conference on [railsdog](#) as well as here at the End Point blog. Finally, the [SpreeCommerce](#) site will be updated with various Spree related announcements.

## Google Base and Spree

Several clients I've worked with have integrated [Google Base](#). I've always wondered how much Google Base actually impacts traffic without having access to the data.

After the recent development of a [google base spree extension](#), we've been able to measure the significant impact of search engine traffic directly related to Google Base. The extension was installed for one of our [Spree](#) clients. See the data below:



Search engine traffic (y-axis is hits where the referring site was a search engine). Google Base was installed on March 24th.



Other referral traffic during the same time period (y-axis is hits where the referring site was not a search engine and traffic was not direct).

Although the extension is relatively simple and has room for improvement (missing product brands, product quantity), you can see how much traffic was impacted. Hopefully with future extension improvements, traffic will increase even more.

Learn more about End Point's [SEO and analytics services](#).

## Git commits per contributor one-liner

Just for fun, in the [Spree Git repository](#):

```
git log | grep ^Author: | sed 's/ <.*//; s/^Author: //' | sort | uniq -c | sort -nr
813 Sean Schofield
97 Brian Quinn
81 Stephanie Powell
42 Jorge Calás Lozano
37 paulcc
27 Edmundo Valle Neto
16 Dale Hofkens
13 Gregg Pollack
12 Sonny Cook
11 Bobby Santiago
8 Paul Saieg
7 Robert Kuhr
6 pierre
6 mjwall
6 Eric Budd
5 Fabio Akita
5 Ben Marini
4 Tor Hovland
4 Jason Seifer
2 Wynn Netherland
2 Will Emerson
2 spariev
2 ron
2 Ricardo Shiota Yasuda
1 Yves Dufour
1 yitzhakbg
1 unknown
1 Tomasz Mazur
1 tom
1 Peter Berkenbosch
1 Nate Murray
1 mwestover
1 Manuel Stuefer
1 Joshua Nussbaum
1 Jon Jensen
1 Chris Gaskett
1 Caius Durling
1 Bernd Ahlers
```

## Spree 0.4.0 Released

Spree 0.4.0 was officially released today. Spree is a complete open source ecommerce platform written for Ruby on Rails. While Spree technically works "out of the box" as a fully functional store, it is really intended to serve as a strong foundation for a custom commerce solution. Like Rails, Spree is considered to be "[opinionated software](#)", and it does not seek to solve 100% of the commerce needs of all possible clients. Developers are able to provide the missing functionality by using the powerful [extension system](#).

The current release of Spree contains many significant improvements from the previous 0.2.0 release. Some of the highlights include:

- Rails 2.1 support
- SEO improvements
- Security enhancements
- Public assets for extensions
- Mailer templates for extensions
- VAT inclusive pricing

- Taxonomy

Most open source projects in the Rails space are maintained by a single individual and tend to be limited in scope. For Spree we seek to create a large and healthy open source community similar to the ones found in more mature languages and frameworks. The Spree project has received contributions from over twenty different developers and has been translated into five additional languages.

I created Spree in late 2007, and End Point later became an official sponsor of the Spree project and employs me and several other Spree contributors.

Tel: +1 212-929-6923  
Tollfree: +1 888-351-3239  
Fax: +1 212-929-6927

© 2011 End Point Corporation  
920 Broadway Suite 701  
New York, NY 10010 USA



<a href="#">contact</a>	<a href="#">services</a>
<a href="#">clients</a>	<a href="#">team</a>
<a href="#">blog</a>	<a href="#">sitemap</a>