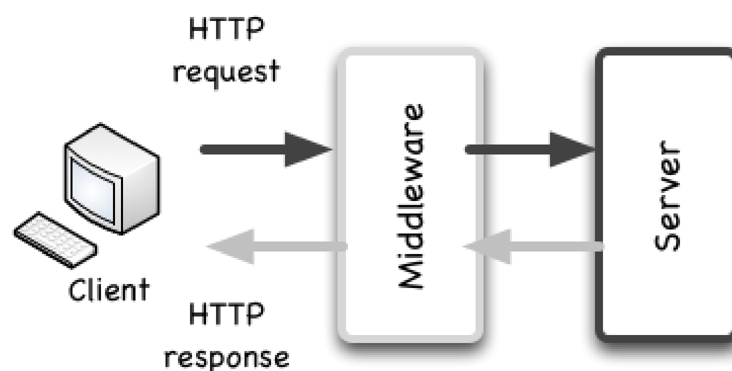
Writing modular web applications with Rack

If you have worked with web applications in Ruby in the past few years, you might have heard of a web-based interface library called Rack. If you haven't, you might want to look under the hoods of your web framework, you'll likely find it nestled comfortably somewhere inside its core. The premise of Rack is simple – it just allows you to easily deal with HTTP requests.

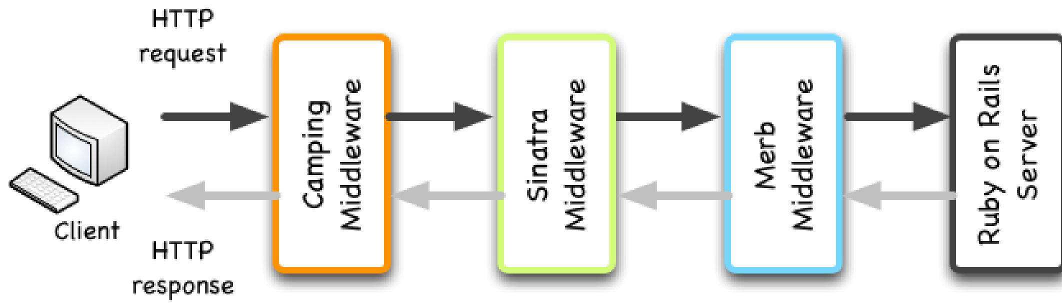
HTTP is a simple protocol; it just basically describes the activity of a client sending a HTTP request to a server and the server returning a HTTP response. Both HTTP request and response in turn have very similar structures. A HTTP request is a triplet consisting of a method and resource pair, a set of headers and an optional body while a HTTP response is triplet consisting of a response code, a set of headers and an optional body.

Rack maps closely to this. A Rack application is a Ruby object that has a `call` method, which has a single argument, the environment, (corresponding to a HTTP request) and returns an array of 3 elements, status, headers and body (corresponding to a HTTP response). This simple definition allows it to be used as a foundational interface for building more sophisticated web frameworks such as Ruby on Rails, Merb and Sinatra.

Besides being the basic building block for most Ruby web frameworks, Rack enables a very interesting feature, somewhat appropriately called 'middleware'. The fundamental idea behind Rack middleware is come between the calling client and the server, processing the HTTP request before sending it to the server, and processing the HTTP response before returning it to the client.



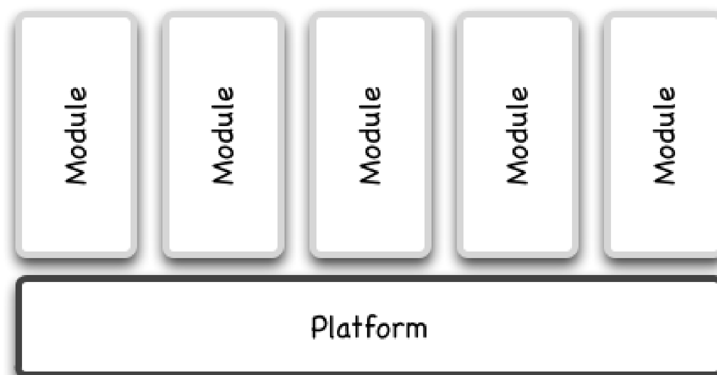
What makes this idea especially interesting and useful is that middleware can be stacked on top of each other! Coupled with the fact that many of the Ruby web frameworks out there are built on top of Rack and therefore any applications built with those frameworks can be Rack middleware, we can build really modular applications with a number of combinations!



So how is this useful? One very practical example is developing large web applications in distributed teams. We can break down the application into sets of features, each implemented in a different piece of middleware, each with its own set of models, views and controllers. Integration is then just taking each piece of the middleware and stacking them up in the main server, which can be a very basic web application (and in fact, is another piece of middleware!).

In this post, I'm going to show a very simple example of how this can be done. I will be using Sinatra for both the middleware as well as the server. All the code in this example is in http://github.com/sausheong/modular. Note that this sample is very Sinatra specific – each Rack-based web framework would likely to have some variance in the way this is implemented. Let's look at a use case first.

You are a project manager for a new project and you have been given the unenviable task of delivering a project with a tight deadline. In order to help you with this, your management has given you a sizable team to do the job. That's the good news. The bad news is that the people in the project team are distributed around the world (since they are formed from various existing teams). The best way to do this, you decided, is to split up project by modular, independent features and get the various team members to deliver them in parallel. You also decided that you needed a core few team members to build the common functions and the platform upon which the other team members build their modules on. The architecture looks something like this.



Let's look at the implementation of the application. First, we have a base server application, which is written by the platform team. The base server application provides the framework for the rest of the modules to sit on. It

should provide a number of fundamental services, but in this example, I will only show simple user authentication and a set of common helper functions.

```
require'rubygems'
%w(haml sinatra rack-flash json rest_client
active_support).each  { |gem| requiregem}
%w(user).each {|model| require model}
%w(sinatra/common_helper middleware).each {|feature| require
feature}

set :sessions, true
set :show_exceptions, false
use Rack::Flash
use Middleware::App

get '/'do
  redirect '/dashboard'if session[:id]
  redirect '/login'
end

get '/login'do haml :login, :layout =>falseend

post '/login'do
if authenticate(params[:email], params[:password])
    redirect '/'
else
    redirect_with_message '/login', 'Email or password wrong.
Please try again'
end
end

get '/logout'do
  session.clear
  redirect '/'
end

get '/dashboard'do
  require_login
  haml :dashboard
end


error do  redirect '/'end
```

http://github.com/sausheong/modular/blob/master/server/server.rb

In the `server` folder of repository, you will find a normal Sinatra application. In fact this forms a skeletal frame of a very simple web application with user authentication capabilities. Let's look at some of the code.

```ruby
%w(user).each {|model| require model}
```

This requires the user model, defined in DataMapper. You will not find it in the `server` folder because it is in a separate folder named `server-models`. If you look into the `server-models` folder you will find that it is set up to be packaged as a gem. Why is this so? This is because the user data model is a part of the base platform, used in most parts of the application, including most if not all other modules. In order to allow other team members to have access to the user data model, we package it as a gem and deliver it through a private gem server (more about this later). For now, just notice that we deploy the user data model as a gem, and require it in the server application.

```ruby
%w(sinatra/common_helper middleware).each {|feature| require
feature}
```

Next, we require two other packages. If you look at the other top-level folders in the source code, you'll realize that both these packages are also gems, and as you would have guessed it, they are delivered to the server through a private gem server and required in the server application as well.

The `CommonHelper` package is as its name implies, a set of commonly used Sinatra helpers. In the Sinatra documentation, they are known as Sinatra Extensions.

The `Middleware` package is the Rack middleware package we will inspect in depth in a while, also packaged as a gem. Note that requiring it is not enough, in a couple of lines further down, you need to use it as well.

```ruby
use Middleware::App
```

The rest of the code is relatively simple so I'll skip them.

Let's look at the user data model next. Again, this is rather straightforward DataMapper code, nothing fanciful.

```ruby
require'dm-core'
require'dm-migrations'

DataMapper.setup(:default,
'mysql://root:root@localhost/modular')

classUser
includeDataMapper::Resource

  property :id, Serial
  property :name, String
  property :email, String
end
```

http://github.com/sausheong/modular/blob/master/server-models/lib/user.rb

You might notice that this and all other packages are distributed as gems. This is really for the convenience of distributing changes to the rest of the team. Essentially, whatever changes you make to your own packages, you simply push a new version of the gem to the private gem server and let the rest of the team know. For the rest of the team, this works well because they can choose to use the latest version or use a prior version in case the latest doesn't work properly.

Now let's look at the common helper.

```
require'sinatra/base'

moduleSinatra
moduleCommonHelper

defrequire_login
      redirect_with_message('/login', 'Please login first')
unless session[:id]
end

defauthenticate(email, password)
      response
=RestClient.post('https://www.google.com/accounts/ClientLogin'
,
'accountType' =>'HOSTED_OR_GOOGLE',
'Email'  => email,
'Passwd' => password,
:service =>'xapi',
:source  =>'Goog-Auth-1.0') do |response, request, result,
&block|

        user =User.first :email => email
if response.code ==200andnot user.nil?
          session[:id] = response.to_s
          session[:user] = user.id.to_s
returntrue
end
returnfalse
end
end

defredirect_with_message(to_location, message)
      flash[:message] = message
      redirect to_location
end
```

```
end
  helpers CommonHelper
end
```

http://github.com/sausheong/modular/blob/master/common_helper/lib/sinatra/common_helper.rb

As I mentioned earlier, this is actually a Sinatra extension, packaged in a gem. Note that we require `sinatra/base` instead of `sinatra` here. In this example, I used Google's ClientLogin as the authentication mechanism, because of the relative ease of using it. Under actual production conditions, you will want to use something like OAuth instead.

Notice this line near the bottom of the code:

```
helpers CommonHelper
```

This essentially registers these helpers in whichever Sinatra application or middleware that is requiring in this module.

Until now, we've been only dealing with code that is written by the platform team members. These are the packages that will be used by other module team members and they are distributed as gems through a private gem server. Now let's look at the code the module teams will be developing.

Let's start with the data model used in a module. This is found in the middleware-models folder of the sample code.

```
require'dm-core'
require'dm-migrations'
require'user'

DataMapper.setup(:default,
'mysql://root:root@localhost/modular')

classOrder
includeDataMapper::Resource

  property :id, Serial
  property :created_on, DateTime
  property :order_no, String

belongs_to:user
end
```

http://github.com/sausheong/modular/blob/master/middleware-models/lib/order.rb

As in the user data model, this is simply DataMapper stuff. Take note though that we refer to the user model here.

Now let's look at the middleware itself.

```
require'sinatra/base'
require'haml'
require'order'
require'sinatra/common_helper'

moduleMiddleware
classApp< Sinatra::Base
    helpers Sinatra::CommonHelper

    configure do
      set :public, File.join(File.dirname(__FILE__), '..',
'public')
      set :views, File.join(File.dirname(__FILE__), '..',
'views')
      set :server_layout,
File.read(File.join(File.dirname(File.expand_path($0)),
'views/layout.haml'))
end

    get '/orders'do
      require_login
@orders=Order.all
      haml :list, :layout => settings.server_layout
end

    get '/orders/:id'do
@order=Order.get params[:id]
      haml :show, :layout => settings.server_layout
end

end
end
```

http://github.com/sausheong/modular/blob/master/middleware/lib/middleware.rb

Let's go into details in this code. Firstly, note that this is implemented as a Sinatra application in the non-classic way, that is, we sub-classed `Sinatra::Base`. This is because the classic style pollutes the namespace and assumes a single view and public folder, making it difficult to stack multiple Sinatra applications like we want to.

Next, because we sub-class `Sinatra::Base`, we need to explicitly register `CommonHelper` as a helper. Notice that because `CommonHelper` is a Sinatra extension, we can use it in the server as well as in any of the middleware.

Next, we set some configurations.

```
set :public, File.join(File.dirname(__FILE__), '..', 'public')
set :views, File.join(File.dirname(__FILE__), '..', 'views')
```

The two settings for public and view folders are necessary because otherwise, we'll fall back on the server's public and view folders, which mean we will not have the clean separation we want. Remember that each module is supposed to be independent and self-contained – we want to encapsulate all views and other assets used only in this module.

```
set :server_layout,
File.read(File.join(File.dirname(File.expand_path($0)),
'views/layout.haml'))
```

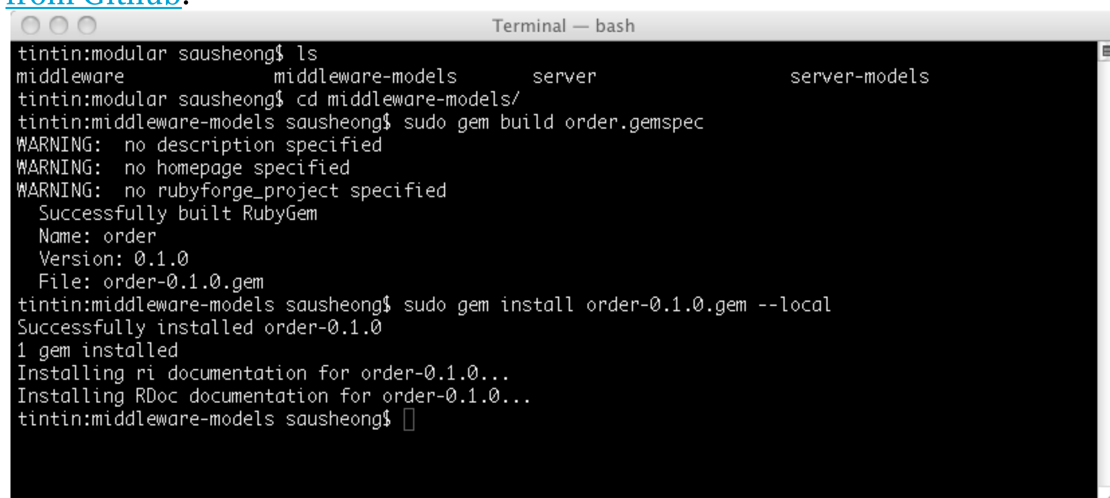
The next setting is interesting because while we want the modules to be independent, we don't want to copy the layout view into every piece of middleware we create, and update them every time something changes at the server. This configuration essentially searches for the layout file in the server (here I'm assuming the view folder is named `view` and we're using Haml) and loads it up in the middleware. This layout (in Haml) will be used when we specify which layout to use in the routes.

```
haml :list, :layout => settings.server_layout
```

Note that the `view` folder is bundled together as part of the gem. This is what we want, because we want this team to be wholly responsible for the feature.

We've gone through the code quickly; now let's see how it is used. I will talk about how we can deploy this step by step.

First, we need to create the models. In the screenshots below, I built the gems using the user and order data models. Firstly, you need to download the code from Github.



```
Terminal — bash
tintin:server-models sausheong$ sudo gem build user.gemspec
WARNING:  no description specified
WARNING:  no homepage specified
WARNING:  no rubyforge_project specified
  Successfully built RubyGem
  Name: user
  Version: 0.1.0
  File: user-0.1.0.gem
tintin:server-models sausheong$ sudo gem install user-0.1.0.gem --local
Successfully installed user-0.1.0
1 gem installed
Installing ri documentation for user-0.1.0...
Installing RDoc documentation for user-0.1.0...
tintin:server-models sausheong$
```

After creating the models, I go into irb and use DataMapper to migrate them (this assumes I have already created a MySQL database with the name `modular`). I need to create a user as well. This is only really applicable for the authentication and it can be any Gmail account (since I'm using Google ClientLogin).

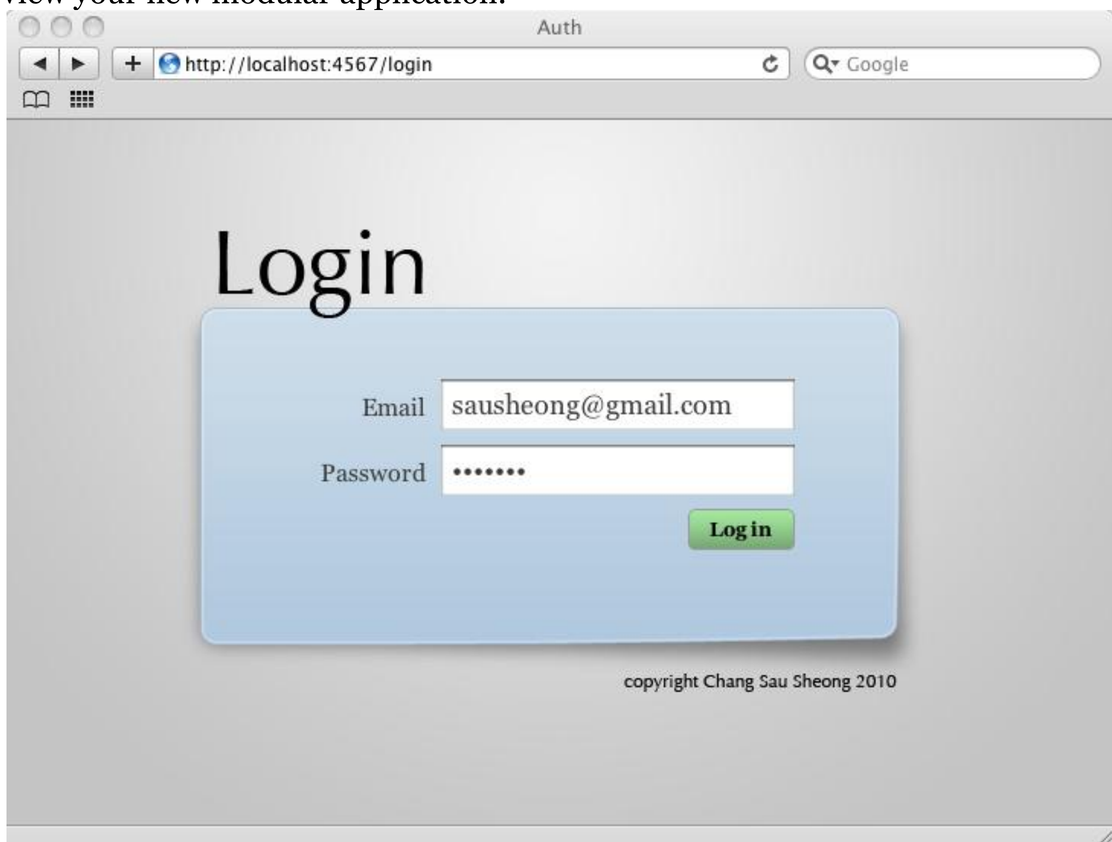
```
Terminal — ruby
tintin:modular sausheong$ irb -r order -r user
>> DataMapper.auto_migrate!
=> #<DataMapper::DescendantSet:0x101ef8060 @descendants=[User, Order]>
>> User.create :name => 'Sau Sheong', :email => 'sausheong@gmail.com'
=> #<User @id=1 @name="Sau Sheong" @email="sausheong@gmail.com">
>>
```

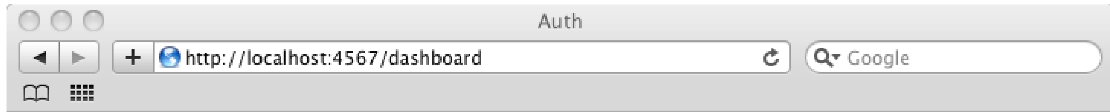
As with the data models, I need to package the common helpers and the middleware into gems as well.

```
Terminal — bash
tintin:common_helper sausheong$ gem build common_helper.gemspec
WARNING:  no description specified
WARNING:  no homepage specified
WARNING:  no rubyforge_project specified
  Successfully built RubyGem
  Name: common_helper
  Version: 0.1.0
  File: common_helper-0.1.0.gem
tintin:common_helper sausheong$ sudo gem install common_helper-0.1.0.gem --local
Password:
Successfully installed common_helper-0.1.0
1 gem installed
Installing ri documentation for common_helper-0.1.0...
Installing RDoc documentation for common_helper-0.1.0...
tintin:common_helper sausheong$
```

```
tintin:modular sausheong$ cd middleware
tintin:middleware sausheong$ gem build middleware.gemspec
WARNING:  no description specified
WARNING:  no homepage specified
WARNING:  no rubyforge_project specified
  Successfully built RubyGem
  Name: middleware
  Version: 0.1.0
  File: middleware-0.1.0.gem
tintin:middleware sausheong$ sudo gem install middleware-0.1.0.gem --local
Successfully installed middleware-0.1.0
1 gem installed
Installing ri documentation for middleware-0.1.0...
Installing RDoc documentation for middleware-0.1.0...
tintin:middleware sausheong$ 
```

And that’s done! Now, run the server application and then fire up a browser to view your new modular application.



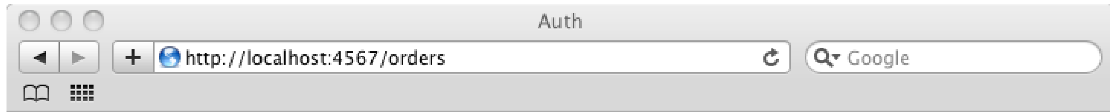


Success!

You have successfully logged in!

Show all orders
Show first order
Logout

Notice here that the layout for the orders page is from the server, since the middleware does not provide any layouts. You can, of course, set layouts for every piece of middleware though most applications would want to keep a consistent user experience.



Building web applications used to be done in smaller and geographically closer teams. Now that web applications are the mainstream, and globally distributed development teams are common, it's important to be able to organize the development in a more modular and effective way. If you're using Ruby in any web application development, I would encourage you to try out Rack to make it more modular!