PeepCode
press

$9

Active Merchant

Show me the money!

by Cody Fauser

RELEVANT!
RAILS
2.0.2
ACTIVEMERCHANT 1.3.1

CONTENTS

Introduction

CHAPTER 1

ActiveMerchant is a payment processing library for Ruby. ActiveMerchant can be used standalone, but also integrates wonderfully into Rails applications as a Rails plugin. Like Rails and many other successful open source projects, ActiveMerchant is an extraction. ActiveMerchant was extracted from Shopify (http://www.shopify.info/?ref=topfunky) in 2005 by Tobias Lütke, the co-founder of Shopify.

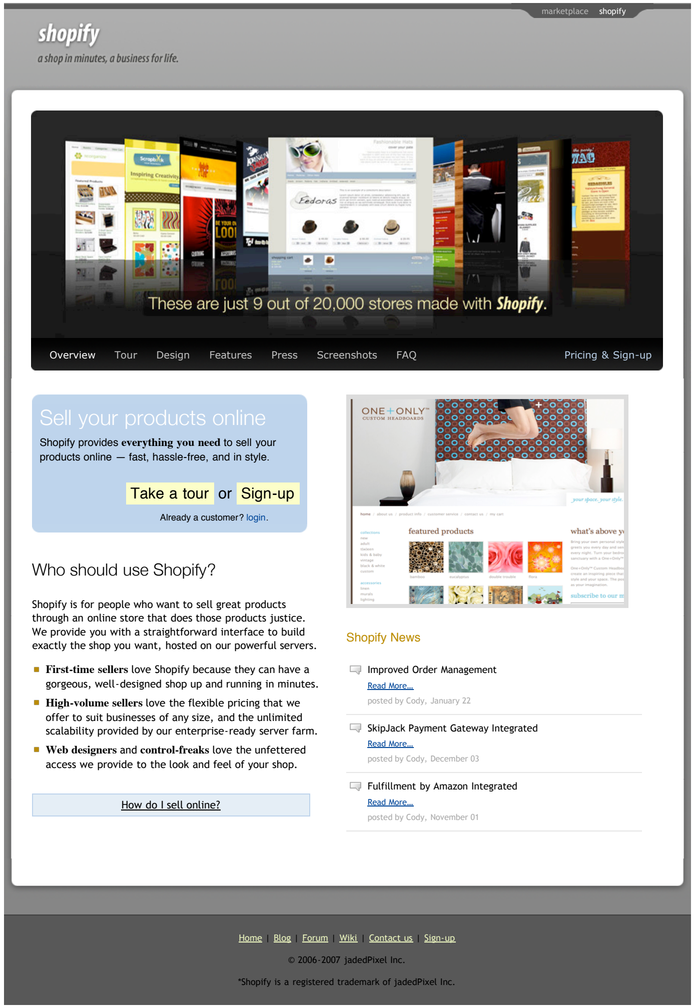
The goal of ActiveMerchant is simplicity. ActiveMerchant strives to provide a common API for all supported gateways. This provides a huge benefit to users of the ActiveMerchant library. Once you've integrated credit card processing into your application once with ActiveMerchant you don't have to worry about learning an entirely new API for your next project. You can also switch payment providers midway through a project with minimal or no changes to your code.

ActiveMerchant is actively under development by Jaded Pixel and its many contributors. New gateways are added regularly. ActiveMerchant now supports over 30 gateways based in countries all over the world.

ActiveMerchant is also very mature project. Since June 2006 it has been used to process millions of dollars of transactions for Shopify customers alone. This is in addition to the many other projects using ActiveMerchant for their payment processing.

ActiveMerchant was designed to make accepting payments in your application as simple and straightforward as possible. Here are a few of the many reasons why you should use ActiveMerchant and not attempt to roll your own solution:

- Under active development.



Sell your products online

Shopify provides **everything you need** to sell your products online — fast, hassle-free, and in style.

Who should use Shopify?

Shopify is for people who want to sell great products through an online store that does those products justice. We provide you with a straightforward interface to build exactly the shop you want, hosted on our powerful servers.

- **First-time sellers** love Shopify because they can have a gorgeous, well-designed shop up and running in minutes.
- **High-volume sellers** love the flexible pricing that we offer to suit businesses of any size, and the unlimited scalability provided by our enterprise-ready server farm.
- **Web designers** and **control-freaks** love the unfettered access we provide to the look and feel of your shop.

Shopify News

Improved Order Management
Read More...
posted by Cody, January 22

SkipJack Payment Gateway Integrated
Read More...
posted by Cody, December 03

Fulfillment by Amazon Integrated
Read More...
posted by Cody, November 01

© 2006-2007 jadedPixel Inc.

*Shopify is a registered trademark of jadedPixel Inc.

- Support for over 30 payment gateways.
- Consistent API between all supported gateways.
- Nicely abstracted so you don't have to worry about underlying gateway implementations.
- Automatic SSL peer verification.
- Built-in retry logic for connection errors.
- Sophisticated validation logic for credit cards
- Well tested and used in production since June 2006 by Shopify (http://www.shopify.com/?ref=topfunky).

The goal of the document is to get you up to speed with ActiveMerchant as quickly as possible. The discussion in this document is limited to authorization & capture of payments and doesn't get into the more complex features that ActiveMerchant offers such as recurring billing, reference purchases, and storing credit cards in a processor's vault systems. However, by the end of the document you'll have the working knowledge to easily master these more complicated scenarios with ActiveMerchant. The document is split up into several sections.

This introductory section builds the foundation for the rest of the document. It starts off with some background on ActiveMerchant, goes through installation of the project, and finishes off with the development of an example where you'll perform test purchase with a real payment gateway.

The second section walks through development of a simple sample application. The sample application gives a glimpse of actual working code using ActiveRecord backed models. The section covers many topics including storing transaction data returned from the payment gateway, management of order state, and strategies for testing your application.

The third section introduces the Payment Card Industry Data Security Standard (PCI DSS) and then goes over some of the security precautions you should take in your applications to help meet the standard. The precautions include encrypting the transmission of cardholder data using SSL and preventing accidental leaks or storage of customer card holder data in your application's logging facilities.

Terminology

This document uses a lot of terms and concepts from the payment processing industry. The next chapter gives an overview of some of the fundamental concepts of payment processing. An overview of the entire e-commerce ecosystem can be found in the article Credit Card Processing: How It All Works (http://www.practicalecommerce.com/articles/168/Credit-Card-Processing-How-It-All-Works/) at Practical eCommerce (http://www.practicalecommerce.com). For comprehensive list of payment processing terms you can take a look at Authorize.net Glossary (http://www.authorize.net/resources/glossary/).

Processing Models

Payment gateways provide both the ability to accept and refund payments from a customer's credit cards. The following is a brief overview of the entire process that occurs from the time the customer submits their credit card information to the time that the authorized funds are deposited into the merchant's account.

A merchant account is a special type of bank account, issued by a merchant bank, that allows the merchant to accept credit card payments. The payment gateway acts as a gateway to the merchant bank's processor which itself acts as a gateway to the Credit Card Interchange. The Credit Card Interchange is a network of financial institutions which communicate with each other to manage the processing and clearing of credit card transactions. The Credit Card Interchange then authorizes the amount of the purchase with the issuer of the cardholder's credit card. The result of the transaction is then transmitted all the way back to the payment gateway and to the merchant. In our case the result is returned to ActiveMerchant gateway making the API request. The entire process is typically completed in under 3 seconds. Authorization is initiated by a payment gateway's `authorize()` method in ActiveMerchant.

Transferring the funds to the merchant's account, called settlement, follows a similar flow. The payment gateway instructs the Credit Card Interchange via the merchant bank's processor to transfer the authorized funds to the merchant's bank and ultimately to the merchant's bank account. Instructing the payment gateway to settle a transaction is initiated by a payment gateway's `capture()` method in Active-Merchant.

A great visual overview of the entire process is the Authorize.net How it Works diagram (http://www.authorize.net/resources/howitworksdiagram/).

As you can see from the process described above, the payment gateway doesn't actually receive any of the money transfered from the customer's credit card. The payment gateway is an entry point into the entire credit card processing system that enables the merchant to process credit cards. Next we're going to go into more detail about accepting and refunding payment.

FINDING A MERCHANT ACCOUNT

When we were looking for a merchant account, we started by talking to the bank where our other business checking accounts were located. It seemed that it would make it easier to transfer funds if all our accounts were handled by the same bank.

Unfortunately, the bank had fees and restrictions that weren't hospitable to the size of our business at the time. It turned out to be easier to just use the merchant account associated with our payment gateway.

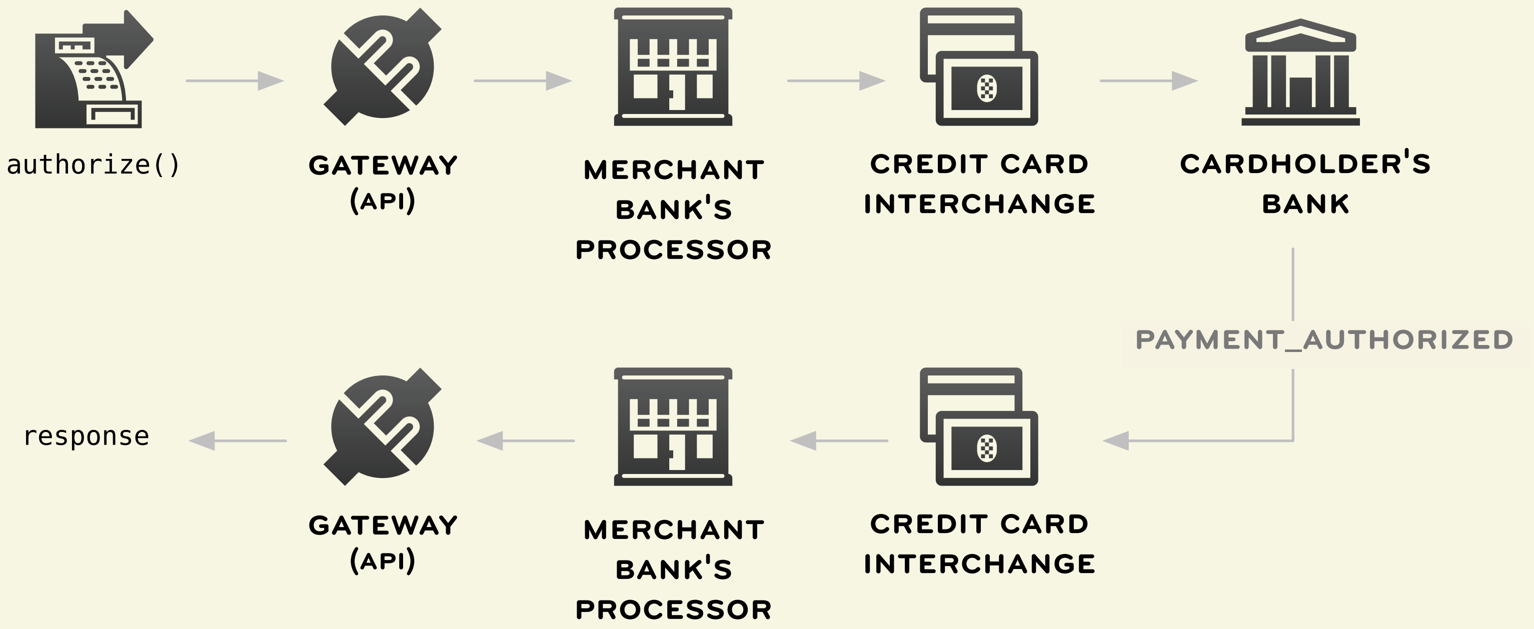
The money ends up in our accounts eventually and it was less hassle to do a daily transfer from the third-party merchant account than it would have been to link the payment gateway to our main bank.

—PJ Hyett, Err Free (http://errfree.com)

Accepting payment

Normally, credit card payments occur in two distinct steps called authorization and capture. The first step is the authorization of the amount of the purchase against the buyer's credit card. An authorization guarantees that funds for the amount of the purchase are available on the buyer's card. Usually an authorization remains open

PAYMENT AUTHORIZATION



authorize()

GATEWAY (API)

MERCHANT BANK'S PROCESSOR

CREDIT CARD INTERCHANGE

CARDHOLDER'S BANK

PAYMENT_AUTHORIZED

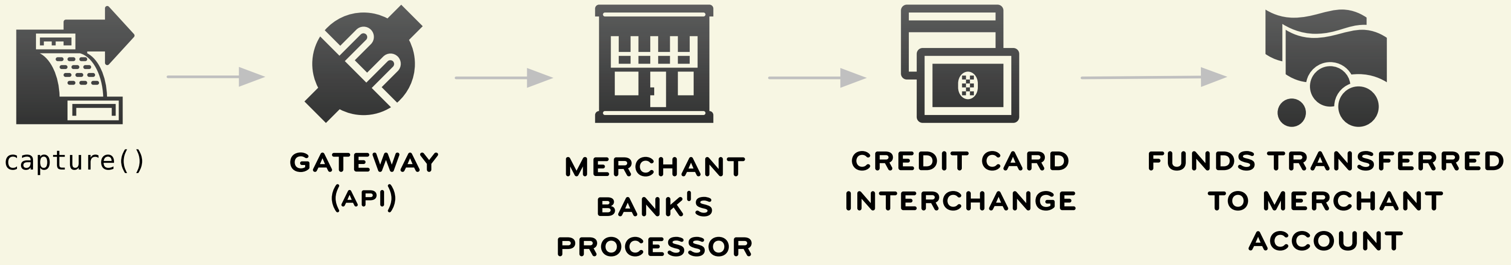
CREDIT CARD INTERCHANGE

MERCHANT BANK'S PROCESSOR

GATEWAY (API)

response

SETTLEMENT



capture()

GATEWAY (API)

MERCHANT BANK'S PROCESSOR

CREDIT CARD INTERCHANGE

FUNDS TRANSFERRED TO MERCHANT ACCOUNT

for around 30 days, but the funds may only be guaranteed by the payment gateway for up to 7 days.

The second step is the capture. The capture indicates to the payment gateway that the authorized funds are to be settled. Settlement is the process that actually transfers the authorized funds to the merchant. Usually, all captured authorizations are settled once daily in a batch by the payment gateway.

The process has been separated into two distinct parts to facilitate the sale of physical goods where the fulfillment of goods does not happen immediately. A benefit of having the authorization and capture as distinct steps is that most gateways allow for capturing anywhere from 0% to 115% of the initial authorized amount. This permits last minute changes in the order such as a change in the shipping method, or a partial refund. It is recommended that authorizations are not captured until the physical goods sold have been shipped to the buyer.

Having distinct steps for authorization and capture works great for the sale of physical goods, but what happens if the goods being sold aren't physical, or the goods are fulfilled immediately? This is where the concept of a purchase comes in. The purchase is essentially both an authorization and a capture done in a single step. A purchase may also be referred to as a sale.

ActiveMerchant gateways provide methods for both `authorization()` and `purchase()`, but only if the payment gateway supports the functionality. Both methods take the same arguments, so it is easy to switch between the two different models for accepting payment in your application.

Refunding payment

Sometimes it is necessary to cancel an authorization or purchase, or to return funds to a customer. Cancelling an unsettled authorization or purchase is called a void. Returning funds to a customer's credit card is called a credit.

Voiding prevents an authorization or purchase from ever being settled by the payment gateway. This means that a void can only be performed on an authorization, or purchase which has not yet been settled by the payment gateway. Voiding occurs before any funds are actually transferred from the customer to the merchant and is the best way to cancel an order. A credit must be performed if the funds have already been settled.

A credit transfers funds back to a customer's credit card. Credits can be done for the entire amount of a sale or for less than the original amount. Crediting less than the original amount is called a partial credit.

There are also two different ways to credit funds back to a customer's credit card. The first way is called a referenced credit. A referenced credit does not involve sending the customer's credit card information to the payment gateway a second time. Instead a reference to the initial authorization, or purchase is sent to the payment gateway.

If there is no initial transaction that can be referenced then there is also the option of performing a non-referenced credit. A non-referenced credit sends the customer's credit card information to the payment gateway and is essentially a purchase done in reverse. Non-referenced credits are not common and are not offered by all payment gateways. Additionally, most payment gateways do not enable non-referenced credits by default in a merchant's account. In most cases the merchant must specifically enable the feature in their account profile with the payment gateway.

Disabling non-referenced credits is the most secure configuration for a merchant's payment gateway account. Without non-referenced credits it is more difficult to perform fraud using the merchant's account because an initial transaction must be referenced for every credit.

Charging Credit Cards

CHAPTER 2

We can start charging credit cards now that ActiveMerchant has been installed, and we know a bit about credit card purchasing models. In this section we're going to run through a basic credit card purchase, and then enhance the basic purchase code to include additional customer and order information in the request to the payment gateway. By the end of this section you'll have completed some test transactions, and you'll have a good grasp on the fundamental concepts of ActiveMerchant.

The Braintree Payment Gateway (http://www.braintreepaymentsolutions.com) was chosen for this document because it offers a full feature set and generic test account credentials. It also offers features beyond the topics covered in this document, such as recurring billing and secure customer data storage. The Braintree API Documentation is available on request from Braintree (http://www.braintreepaymentsolutions.com/contact.php). It contains the test account information and test credit card numbers used throughout this document.

You can also use any other payment gateway supported by ActiveMerchant instead of the Braintree gateway. You just need to substitute its class name for `BraintreeGateway` throughout this document. There are slight variations in the feature set of each gateway, so expect slight variations from the results shown. The ActiveMerchant RDOC Documentation (http://activemerchant.rubyforge.org/) has a complete list of all supported payment gateways.

Installation

ActiveMerchant is very easy to install. Following are some simple instructions for installing ActiveMerchant as a Ruby Gem, or as a

Rails plugin. This document assumes that you are using ActiveMerchant version 1.3.1 or higher and at least Rails version 2.0.2.

Rails Plugin

Installation as a Rails plugin has the advantage that you don't explicitly require ActiveMerchant anywhere in your code. Rails will automatically load ActiveMerchant on startup because ActiveMerchant provides an `init.rb` file.

Installation as a Rails plugin is also advantageous if you wish to run ActiveMerchant on the bleeding edge because you can use SVN externals to track the latest version of the trunk.

Run the following command from the root directory of your Rails project to install ActiveMerchant as a Rails plugin:

```
$ ./script/plugin install http://activemerchant.googlecode.
com/svn/trunk/active_merchant
```

RUBY 1.8.6

If you encounter errors while running the examples, you may need to upgrade to Ruby 1.8.6.

We received odd card validation and login errors with Ruby 1.8.4, but running the code on a machine with Ruby 1.8.6 worked fine.

Mac OS X ships with a current version of Ruby at `/usr/bin/ruby`.

For Windows, try the One-Click Ruby Installer (http://rubyforge.org/projects/rubyinstaller) or Instant Rails (http://rubyforge.org/projects/instantrails).

Linux users can download the latest source from ruby-lang.org (http://www.ruby-lang.org/en/downloads). You may also need the readline (http://tiswww.case.edu/php/chet/readline/rltop.html) library for using Ruby's interactive shell.

Ruby Gem

Installation as a Ruby Gem is also very straight forward.

```
$ gem install activemerchant
```

If you are using ActiveMerchant as a Gem in your Rails project then you will have to remember to require the ActiveMerchant Ruby Gem. Add the following statement to your `environment.rb` file:

```
require 'active_merchant'
```

A Simple Purchase

Let's start out by writing the least amount of code that will charge a credit card successfully.

> You can try this out in a single Ruby file. See `examples/initial_purchase.rb` in the sample code files.

First we need to tell ActiveMerchant to use the payment gateway's test server. By default, `ActiveMerchant::Billing::Base.mode = :production`, which means that ActiveMerchant will send all requests to the payment gateway's production server. Set the gateway to `:test` mode as follows:

```
ActiveMerchant::Billing::Base.mode = :test
```

Now we need to construct an instance of the `BraintreeGateway` using the credentials found in the BraintreeGateway API.

examples/initial_purchase.rb

```
gateway = ActiveMerchant::Billing::BraintreeGateway.new({
  :login    => 'demo',
  :password => 'password'
})
```

Notice that all ActiveMerchant gateways are in the `ActiveMerchant::Billing` namespace. This is to provide room for future expansion of the ActiveMerchant library to support other e-commerce services such as fulfillment and shipping services.

Next we need to construct an `ActiveMerchant::Billing::CreditCard` object which will be the payment source for the test transaction. The `CreditCard` class provided by ActiveMerchant mimics an ActiveRecord object for the purposes of validation and construction. The difference is that the ActiveMerchant `CreditCard` objects can not persisted to a database. You can call `valid?` on the credit card. Any resulting validation errors are accessible in the same manner as an ActiveRecord object. The object is also compatible with the `error_messages_for()` Rails view helper.

examples/initial_purchase.rb

```ruby
credit_card = ActiveMerchant::Billing::CreditCard.new({
  :first_name => 'Cody',
  :last_name  => 'Fauser',
  :number     => '4111111111111111',
  :month      => '10',
  :year       => (Time.now.year + 1).to_s,
  :verification_value => '999'
})
```

The `CreditCard` object is constructed with all of the information required by the payment gateway to authorize payment. All of the required information can be found right on the credit card itself. Most of the information is on the front of the card, except for the verification value, which is usually found on the back of the card.

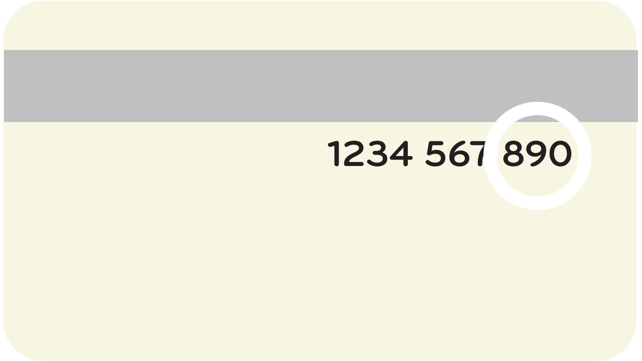
> Never use real credit card information for test transactions. Many test accounts are publicly available and shared between users. Also, the same level of data security may not be present in the test environment as exists in the payment gateway's production environment.

The first two keys, `:first_name` and `:last_name`, we're passing in are the card holder's first name and last name respectively. The name is passed in as it is printed on the card. In this case I'm using my own name, since the card data is purely fictional anyway.

Following is the `:number`, which is the card number printed on the front of the card. We've selected one of the credit card numbers from the Braintree API documentation which is valid for the Braintree test environment. This will ensure that the test purchase succeeds. Some payment gateways have a list of supported test card numbers available in their API documentation, but others will take any valid card number for the type card being used.

The next keys, `:month` and `:year`, are the components of the credit card's expiration date. The `month` is expected as a one or two digit month number in string format. The `year` is expected as a four digit year number in string format. We've set the expiration date to a date in the future so that the purchase will be successful.

Last is the `:verification_value` key. The `verification_value`, also known as the CID, CVC2, or CVV2, is a 3 or 4 digit security code printed on the back of the card. The verification value helps cut down on fraud because it indicates that the customer has physical access to the card. The reason this works is because it is forbidden by the Payment Card Industry Data Security Standard for anyone to store the verification value. This means that merchants requiring the verification value are protected from the use of stolen card numbers, except when the cards have been physically stolen.



> The credit card verification value is required by default, as this is the most secure configuration. However, if you want to accept credit cards without the verification value you can by setting `ActiveMerchant::Billing::CreditCard.require_verification_value = false`.

Next, we validate the credit card, and if the card is valid, perform the purchase by sending the card holder's card data, along with the amount of the purchase, to the payment gateway. ActiveMerchant provides sophisticated credit card validation. Ensuring that the credit card is valid before making the remote call can save on processing fees and make your application much more responsive. There is no reason to send the card data to the payment gateway if the transaction can't possibly be successful.

examples/initial_purchase.rb

```ruby
if credit_card.valid?
  response = gateway.purchase(100, credit_card)

  print "(TEST) " if response.test?

  if response.success?
    puts "The transaction was successful! The authorization
is #{response.authorization}"
  else
    puts "The transaction was unsuccessful because
#{response.message}"
  end
else
  puts "The credit card is invalid"
end
```

We passed in an amount of $1.00, which ActiveMerchant accepts as an integer value in cents. ActiveMerchant processes all amounts as integer values in cents. This prevents any issues arising from the inexact float representation of decimal numbers.

ActiveMerchant then returns the `response` from the payment gateway in an `ActiveMerchant::Billing::Response` object. The `Response` object contains all of the information returned by the payment gateway.

There are several important instance methods on the `Response` object including:

- `success?` - Indicates whether or not the transaction was successful.
- `authorization` - The reference number, provided by the payment gateway, for the transaction.
- `message` - The message returned by the payment gateway.

- test? - Whether or not the transaction was a test transaction.
- cvv_result - Hash containing the code and message from the card company's card verification code check.
- avs_result - Hash result of Address Verification Service results.

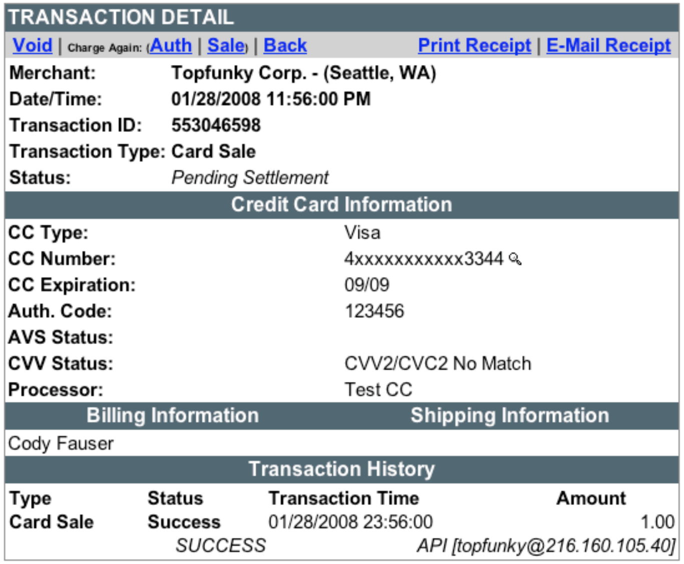
Running the code for the initial purchase should be successful and output something like the following:

```
# => (TEST) The transaction was successful! The authorization
is 3459652
```

Passing the returned Response object instance, response, to the Ruby method pp() is a good way to take a full look at all the data returned by the payment gateway. pp() uses the PrettyPrint library and you must require 'pp' in order to use it. If you're viewing the response in a Rails view you can also use the debug() ActionView helper.

The transaction details are also available in the Braintree admin interface. The credentials, provided in the API reference, are as follows:

- username - demo
- password - password



TRANSACTION DETAIL

Void | Charge Again: (Auth | Sale) | Back   Print Receipt | E-Mail Receipt

Merchant:	Topfunky Corp. - (Seattle, WA)
Date/Time:	01/28/2008 11:56:00 PM
Transaction ID:	553046598
Transaction Type:	Card Sale
Status:	*Pending Settlement*

Credit Card Information

CC Type:	Visa
CC Number:	4xxxxxxxxxxx3344
CC Expiration:	09/09
Auth. Code:	123456
AVS Status:	
CVV Status:	CVV2/CVC2 No Match
Processor:	Test CC

Billing Information	Shipping Information
Cody Fauser	

Transaction History

Type	Status	Transaction Time	Amount
Card Sale	Success *SUCCESS*	01/28/2008 23:56:00	1.00 *API [topfunky@216.160.105.40]*

Resources

- Visa Card Security Features (http://www.visa.ca/en/merchant/fraudprevention/cardfeatures.cfm)
- Card Verification Value 2 (http://www.visa.ca/en/merchant/fraudprevention/cvv2.cfm)

Passing Additional Data

Now that we've got the basic purchase working we can enhance the code we've already written to send along additional customer and order information to the payment gateway. Sending along additional transaction data to the payment gateway is useful because it allows for more detailed reporting and searching within the payment gateway's administrative interfaces.

Another reason to send the payment gateway the buyer's billing address is so that the address provided by the buyer can be checked with the Address Verification System (AVS). AVS matches the numeric portions of the address provided by the buyer to the card holder's billing address on file at the credit card company. AVS is very useful for reducing fraud, as the thief would have to know the billing address of the card holder, but can also result in false positives. Any AVS information contained in the response from the payment gateway is contained in the `ActiveMerchant::Billing::Response#avs_result` hash. The `avs_result` hash contains the following keys:

- `code` - The AVS result code for the particular card type.
- `message` - The full message for the result code.
- `street_match` - Code indicating whether or not the numeric portion of the street address matched.
- `postal_match` - Code indicating whether or not the zip/postal code matched.

The possible codes for the `street_match` and `postal_match` are:

- `Y` - Match.
- `N` - No match.
- `X` - Not supported.
- `nil` - No information available, check not performed.

There are quite a few supported options provided by ActiveMerchant. ActiveMerchant does its best to match the option to the appropriate field provided by the payment gateway. ActiveMerchant does not raise any errors if an option is not supported by a payment gateway. This facilitates easy switching between payment gateways. The options are always passed in to the gateway's methods as the final argument in hash format. The full list of options, which may not be supported by all payment gateways, is:

- `:order_id` - The order number.
- `:ip` - The IP address of the customer making the purchase.
- `:customer` - The name, customer number, or other information that identifies the customer.
- `:invoice` - The invoice number.
- `:merchant` - The name or description of the merchant offering the product.
- `:description` - A description of the transaction.
- `:email` - The email address of the customer.
- `:currency` - The currency of the transaction. Only important when you are using a currency that is not the default with a gateway that supports multiple currencies.
- `:billing_address` - A hash containing the billing address of the customer.
- `:shipping_address` - A hash containing the shipping address of the customer.

The `:billing_address` and `:shipping_address` hashes can have the following keys:

- `:name` - The full name of the customer.
- `:company` - The company name of the customer.

- `:address1` - The primary street address of the customer.
- `:address2` - Additional line of address information.
- `:city` - The city of the customer.
- `:state` - The state of the customer. The 2 digit code for US and Canadian addresses. The full name of the state or province for foreign addresses.
- `:country` - The ISO 3166-1-alpha-2 code (http://www.iso.org/iso/country_codes/iso_3166_code_lists/english_country_names_and_code_elements.htm) for the customer.
- `:zip` - The zip or postal code of the customer.
- `:phone` - The phone number of the customer.

> You must pass in the `:address1` and `:zip` keys in the billing address hash if you want to use AVS.

Let's enhance our code to pass in the customer's billing address, as well as a description of the purchase being made:

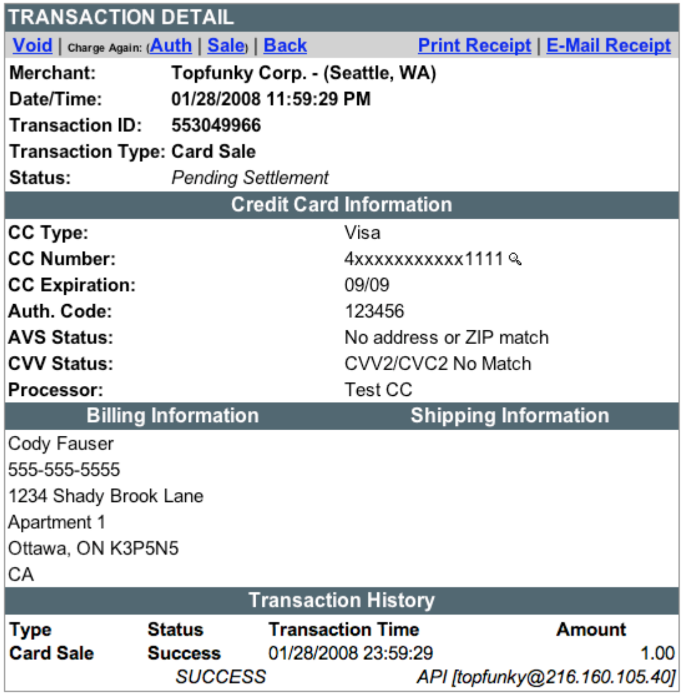
examples/enhanced_purchase.rb

```
options = {
  :billing_address => {
    :name     => 'Cody Fauser',
    :address1 => '1234 Shady Brook Lane',
    :address2 => 'Apartment 1',
    :city     => 'Ottawa',
    :state    => 'ON',
    :country  => 'CA',
    :zip      => 'K3P5N5',
    :phone    => '555-555-5555'
  },
  :description => 'Show me the money!'
}

if credit_card.valid?
  response = gateway.purchase(100, credit_card, options)
```

```
 print "(TEST) " if response.test?

 if response.success?
   puts "The transaction was successful! The authorization
is #{response.authorization}"
 else
   puts "The transaction was unsuccessful because
#{response.message}"
 end
else
 puts "The credit card is invalid"
end
```


Void | Charge Again: (Auth | Sale) | Back
Print Receipt | E-Mail Receipt

TRANSACTION DETAIL

Merchant:	**Topfunky Corp. - (Seattle, WA)**
Date/Time:	**01/28/2008 11:59:29 PM**
Transaction ID:	**553049966**
Transaction Type:	**Card Sale**
Status:	*Pending Settlement*

Credit Card Information

CC Type:	Visa
CC Number:	4xxxxxxxxxxx1111
CC Expiration:	09/09
Auth. Code:	123456
AVS Status:	No address or ZIP match
CVV Status:	CVV2/CVC2 No Match
Processor:	Test CC

Billing Information	**Shipping Information**
Cody Fauser 555-555-5555 1234 Shady Brook Lane Apartment 1 Ottawa, ON K3P5N5 CA	

Transaction History

Type	**Status**	**Transaction Time**	**Amount**
Card Sale	**Success**	01/28/2008 23:59:29	1.00
	SUCCESS		*API [topfunky@216.160.105.40]*

We've added the customer's billing address information, and a short description of the purchase, to this transaction. Viewing the details of this transaction in the Braintree admin interface will now include the customer's billing information.

Now that we've gone through making some test purchases we can move forward with a sample application. The sample application will walk through the development of an application which processes payments and manages order state.

Resources

- Visa E-Commerce Merchant Guide to Risk Management (http://usa.visa.com/download/merchants/visa_risk_management_guide_ecommerce.pdf)
- Address Verification System (http://www.visa.ca/en/merchant/fraudprevention/avs.cfm)

Sample Application

CHAPTER 3

Now that we've taken a quick look at how simple ActiveMerchant is to use we can move on to creating a small application for processing payments. The application will be for my new company Fashionable Money Inc., which sells a variety of fashionable clothing and accessories folded from real currency. The biggest seller is obviously the Fashionable Money Hat™. Let's get started!

Read the history of the moneyhat at the Penny Arcade (http://www.penny-arcade.com/comic/2000/10/23).

Setup and Installation

First, let's create the application. I'm going to call it Moneyhats, but feel free to call yours whatever you like.

```
$ rails moneyhats
```

Now that we have a project, we can enter into it:

```
$ cd moneyhats
```

As of Rails 2.0.2, Rails uses a SQLite3 database by default when generating a new Rails project with the `rails` command. This means that we don't have to worry about creating, or configuring a database, as long as SQLite3 is installed along with the Ruby SQLite3 bindings. Both SQLite3 and the Ruby bindings are installed on Mac OSX Leopard by default. If you're running another OS you'll have to install those packages before proceeding, or else configure a different database such as MySQL.

Installing ActiveMerchant as a plugin to our project is a breeze:

```
$ ./script/plugin install http://activemerchant.googlecode.
com/svn/trunk/active_merchant
```

Since Rails auto-loads all plugins, no require statement is necessary to load ActiveMerchant.

Creating the models

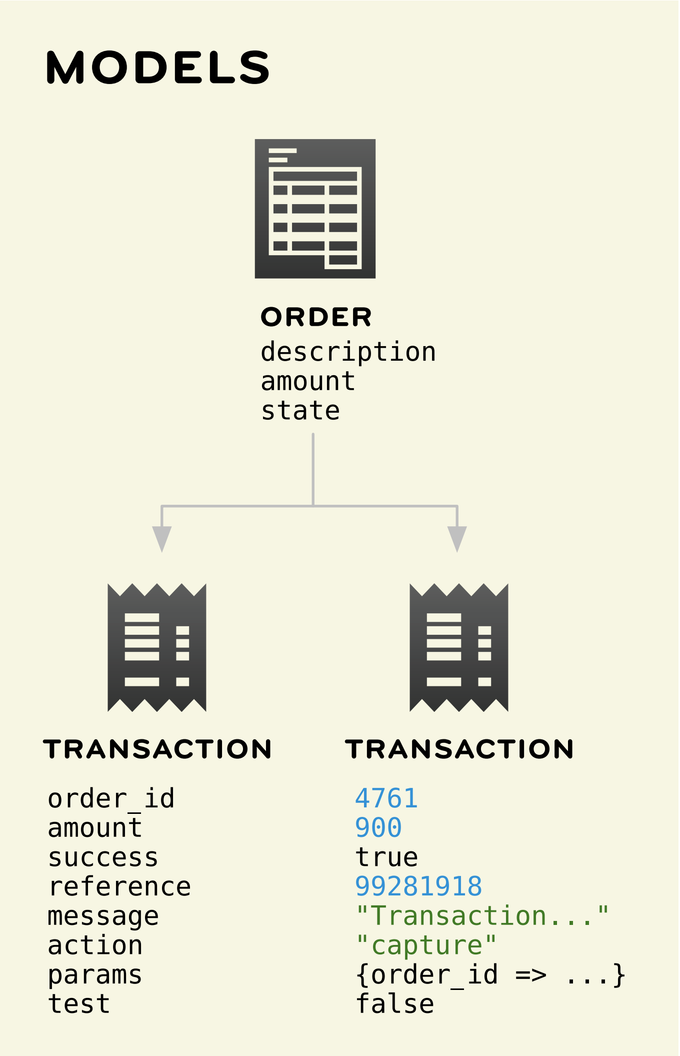
Order Model

We're going to use a very simple `Order` model for the application. The `Order` model will encapsulate the process an order's financial state goes through during its life cycle. Orders will keep a record of all of their transactions using a child collection of `OrderTransaction` objects, which we'll define shortly. Keeping track of an order's state allows the application to intelligently determine which possible gateway operations are applicable for a particular financial state. For example, when an order is in the `authorized` state only the gateway operations `capture` and `void` are applicable to that order.

```
$ ./script/generate model order description:string
amount:integer state:string
```

One thing we need to do is set the default financial `state` of an order to `pending`. This can be done by adding the option `:default => 'pending'` to the column declaration in the migration. This will ensure that all new orders are automatically set to a valid state. Open up `db/migrate/001_create_orders.rb` and set the default `state` to `pending`:

```
application/moneyhats/db/migrate/001_create_orders.rb
class CreateOrders < ActiveRecord::Migration
  def self.up
```



MODELS

ORDER
description
amount
state

TRANSACTION	TRANSACTION
order_id	4761
amount	900
success	true
reference	99281918
message	"Transaction..."
action	"capture"
params	{order_id => ...}
test	false

```
    create_table :orders do |t|
      t.string  :description
      t.integer :amount
      t.string  :state, :default => 'pending'

      t.timestamps
    end
  end

  def self.down
    drop_table :orders
  end
end
```

A brief description of the columns used by the `Order` model:

- `description` - A brief description of the order.
- `amount` - The amount in cents of the payment. The amount is in cents to avoid rounding errors when performing calculations and is the format that ActiveMerchant accepts.
- `state` - The current financial state that the order is in. Default is `pending`.
- `created_at` - The date and time when the order was created.
- `updated_at` - The date and time when the order was last updated.

The `created_at` and `updated_at` columns are both created in the migration with the call `t.timestamps`. Now that the migration for the `Order` model has been updated we can run the migration:

```
$ rake db:migrate
```

Before moving on, let's add the `has_many` association to the `Order` model. Each `Order` object has many `OrderTransaction` objects. Each `OrderTransaction` represents and contains the response data from a single interaction with the payment gateway. Open `app/models/`

order.rb and add the following line of code within the Order class:

```
has_many :transactions,
         :class_name => 'OrderTransaction',
         :dependent => :destroy
```

Notice the association is named transactions instead of order_transactions. This is more aesthetically pleasing and will save a bit of typing when accessing the child collection throughout the code. Since we've strayed slightly from the default Rails naming convention we must use of the :class_name => 'OrderTransaction' option when declaring the association. I also added the :dependent => :destroy option because we don't want orphaned OrderTransactions lingering in the database after an order has been destroyed.

> In a real application, it is highly unlikely that you'd ever want to destroy an order or its related transactions. In order to maintain a complete order history but still allow for deletions, you may want to use the acts_as_paranoid (http://svn.techno-weenie.net/projects/plugins/acts_as_paranoid/) plugin which provides for a soft-delete instead of removing records from the database.

Next we're going to add some code to the OrderTransaction model. We'll get back to adding behavior to the Order model shortly.

OrderTransaction Model

Every interaction with an ActiveMerchant payment gateway returns an instance of an ActiveMerchant::Billing::Response object. We'll use an ActiveRecord backed model to keep a record of all interactions with the payment gateway. It would be nice to call the model Transaction, but this causes problems within Rails because Transaction is a reserved word in Rails. Let's generate a model named

`OrderTransaction` instead. We can specify all of the columns needed for the `OrderTransaction` model when running the generate script:

```
$ ./script/generate model order_transaction order_
id:integer amount:integer success:boolean reference:string
message:string action:string params:text test:boolean
```

The migration created looks as follows:

application/moneyhats/db/migrate/002_create_order_transactions.rb

```
class CreateOrderTransactions < ActiveRecord::Migration
  def self.up
    create_table :order_transactions do |t|
      t.integer :order_id
      t.integer :amount
      t.boolean :success
      t.string  :reference
      t.string  :message
      t.string  :action
      t.text    :params
      t.boolean :test

      t.timestamps
    end
  end

  def self.down
    drop_table :order_transactions
  end
end
```

The columns of the `OrderTransaction` model are straightforward:

- `order_id` - The foreign key reference to the parent order.
- `amount` - The amount in cents of the transaction. The amount is in cents to avoid rounding errors when performing calculations and is the format that ActiveMerchant accepts.

- `success` - Simply a boolean attribute that stores whether or not the transaction was successful.
- `reference` - The identification number or code returned by the payment gateway that allows for reference transactions.
- `message` - The message returned by the payment gateway for the transaction
- `action` - Which type of transaction this represents: authorization, purchase, capture, void, or credit.
- `params` - The response returned by the payment gateway contains a hash of all data pertaining to the transaction. Keeping a record of data is very useful for record keeping and troubleshooting purposes. In this application we're going to use a serialized hash to store this data.
- `test` - A boolean attribute that stores whether or not actual money changed hands during the transaction.
- `created_at` - The date and time when the transaction was created.
- `updated_at` - The date and time when the transaction was last updated.

Let's run the migration for the `OrderTransaction` model:

```
$ rake db:migrate
```

Now that the models and migrations have been defined, we can move on to the more exciting parts of the application which handle the management of order state and payment processing.

Interacting with the Gateway

Let's add some behavior to the `OrderTransaction` model. We're going to define class methods on the model to interact with the Active-

Merchant gateway and return new instances of OrderTransaction objects. Each instance of an OrderTransaction will contain all of the transaction information returned in the ActiveMerchant::Billing::Response from the payment gateway.

Open up app/models/order_transaction.rb and add the completed code for this model:

application/moneyhats/app/models/order_transaction.rb

```
class OrderTransaction < ActiveRecord::Base
  belongs_to :order
  serialize :params
  cattr_accessor :gateway

  class << self
    def authorize(amount, credit_card, options = {})
      process('authorization', amount) do |gw|
        gw.authorize(amount, credit_card, options)
      end
    end

    def capture(amount, authorization, options = {})
      process('capture', amount) do |gw|
        gw.capture(amount, authorization, options)
      end
    end

    private

    def process(action, amount = nil)
      result = OrderTransaction.new
      result.amount = amount
      result.action = action

      begin
        response = yield gateway

        result.success   = response.success?
        result.reference = response.authorization
        result.message   = response.message
        result.params    = response.params
        result.test      = response.test?
      rescue ActiveMerchant::ActiveMerchantError => e
        result.success   = false
        result.reference = nil
        result.message   = e.message
        result.params    = {}
        result.test      = gateway.test?
      end

      result
    end
  end
end
```

First, is the **belongs_to** declaration. The **belongs_to** association is used because each **OrderTransaction** object holds a foreign key reference to its parent **Order** object.

The next line of code is the **serialize** declaration for the **params** column. This declaration allows the entire **Response#params** hash to be stored serialized in the **params** column. The **Response#params** hash contains all of the raw data returned by the payment gateway. However, the keys of the **params** hash returned in a response vary widely between gateways and even between different operations on the same gateway. This is why it is simplest to serialize the entire hash. Keeping as much transaction information as possible is important for investigating failed payments and for troubleshooting problems your application might be having with the payment gateway.

ActiveRecord uses YAML for serialization, which can be slow for very large data sets. Fortunately, the **params** returned from the gateway are usually manageable and won't be retrieved very often.

Next, the **cattr_accessor :gateway** call defines a class accessor named **gateway**. We'll set **gateway** to an instance of the ActiveMerchant gateway that we want to use for processing payments. This accessor allows us to instantiate a different gateway, or a gateway that uses different credentials, in each of our application's environment files: `development.rb`, `test.rb`, and `production.rb`. We'll take a look at the configuration in the next section.

We've now arrived at the most interesting part of the model, which is the actual interaction with the payment gateway. We're defining two class methods, `OrderTransaction.authorize()` and `OrderTransaction.capture()`. Each method returns a new instance of an `OrderTransaction` object containing the response from the payment gateway for the action performed. The model could be easily be extended to support other gateway operations such as purchase, void, and, credit, since all operations return an `ActiveMerchant::Billing::Response` object.

All of the work of creating the new `OrderTransaction` object with the data from the `ActiveMerchant::Billing::Response` is performed by the private class method `process()`. The `process()` method takes the name of the `action` being performed as its first argument and the `amount` of the transaction as its second argument. The action performed is stored so that the different transaction types can be differentiated from one another in the future. A more complex application could even use Single Table Inheritance (STI) with a different subclass for each transaction type.

The `process()` method expects a block and yields the class's ActiveMerchant `gateway` instance to the block. The `process()` method can be used for all types of actions since all transaction methods of ActiveMerchant gateways return an `ActiveMerchant::Billing::Response` object. Additionally, the process action recovers from `ActiveMerchant::ActiveMerchantError` exceptions.

When ActiveMerchant exceptions are rescued, the error data is returned in the `OrderTransaction` object as usual. Fortunately, `Act` `iveMerchant::ActiveMerchantError` exceptions are very infrequent. They are usually caused by refused or dropped connections to the payment gateway's servers. All other exceptions are not rescued because they indicate a more serious problem with the application that is not transient.

Handling the exceptions in the `process()` method prevents us from having to worry about catching the same exceptions in each of the public class methods. Handling the exceptions in a single method also removes the duplication that would occur if the exceptions were handled in each of the public class methods.

Application Configuration

Payment gateways usually provide both a live production server and a test server. The production server is used for live transactions, whereas the test server is merely a simulation. No money is exchanged between the customer and the merchant in the test environment.

> Due to the large number of payment gateways supported by ActiveMerchant it is necessary to check the exact features and environments provided by the gateway you're using in your application.

A slight configuration is required to map the proper gateway environment to each Rails environment and configure the payment gateway we want to use in each environment.

DEVELOPMENT

In development mode we want to use the payment gateway's test environment with the test account credentials:

application/moneyhats/config/environments/development.rb

```
config.after_initialize do
  ActiveMerchant::Billing::Base.mode = :test
end

config.to_prepare do
  OrderTransaction.gateway =
    ActiveMerchant::Billing::BraintreeGateway.new(
      :login    => 'demo',
      :password => 'password'
    )
end
```

The configuration code is placed in the `after_initialize` and `to_prepare` blocks so that it is not executed until Rails has been fully initialized. This ensures that ActiveMerchant and the rest of the Rails application have been loaded before trying to execute the configuration code.

First, we set the mode of ActiveMerchant to `:test` in the `after_initialize` block. This ensures that the payment gateway will send all requests to the gateway's test URL. The `after_initialize` block is executed once after the entire application has been initialized.

Then in the `to_prepare` block we set the `OrderTransaction.gateway` class accessor to the payment gateway we want to use during development. We want to use the `BraintreeGateway` in test mode so that we can simulate real transactions during development. The code needs to be placed in the `to_prepare` block so that it is executed on each reload of the `OrderTransaction` model in development mode. If the code had been placed in the `after_initialize` block then `OrderTransaction.gateway` would be properly set on the first request to the application, but would be `nil` on all subsequent requests.

TEST

In the test environment we don't want to make actual requests to the payment processor. Running the tests would be very slow because of all the remote HTTP requests. It would also be impolite to barrage the gateway's servers with so many test requests. For this reason ActiveMerchant provides the `BogusGateway`. The Bogus gateway is a fake payment gateway that is perfect for use in tests. The following code sets ActiveMerchant to use the `:test` environment and the `OrderTransaction.gateway` to an instance of the `BogusGateway`.

```
application/moneyhats/config/environments/test.rb
config.after_initialize do
  ActiveMerchant::Billing::Base.mode = :test

  OrderTransaction.gateway =
    ActiveMerchant::Billing::BogusGateway.new
end
```

We'll go into more detail about the `BogusGateway` in the following section on testing.

PRODUCTION

Finally, we configure the production environment. You can delay this step until you're ready to launch your application, but I'm showing it for illustrative purposes.

The default ActiveMerchant mode is `:production`, which will send all requests to the payment gateway's live server. In the following configuration we've set it explicitly for the sake of clarity:

```
application/moneyhats/config/environments/production.rb
config.after_initialize do
  ActiveMerchant::Billing::Base.mode = :production

  OrderTransaction.gateway =
    ActiveMerchant::Billing::BraintreeGateway.new(
      :login => 'LIVE_LOGIN',
      :password => 'LIVE_PASSWORD'
    )
end
```

You'll want to replace LIVE_LOGIN and LIVE_PASSWORD with your own credentials provided to you by your payment processor. You may also want to consider loading your live account credentials from an external file that is only available to your application in production. This will ensure that anyone with access to your application code won't also have access to your live gateway credentials.

> Chapter 2 of the PeepCode Code Review PDF (http://peepcode.com/products/draft-rails-code-review-pdf) by Geoffrey Grosenbach walks through a method for using custom configuration files in your Rails application. He also shows how to copy the custom configuration file into your application during deployment.

Testing the OrderTransaction Model

Testing is very important to the success of any project and is even more important when real money is at stake. This section will go through testing the OrderTransaction model both locally with the BogusGateway and remotely with the BraintreeGateway in test mode.

The examples assume you are using Test::Unit but you can use these same concepts in RSpec (http://rspec.info) by adding the appropriate methods to your spec_helper.rb file.

Test Helper

Before we get started with our tests there are a few useful things that can be done to make testing a bit easier. The first is to include the ActiveMerchant::Billing module into the Test::Unit::TestCase class. This will allow us to reference the different ActiveMerchant classes used in the tests with lot fewer key strokes and line noise. Open up test/test_helper.rb and add the following line of code after the class Test::Unit::TestCase declaration:

```
include ActiveMerchant::Billing
```

Two other things that are used frequently in tests are credit cards and addresses. Unit tests will use ActiveMerchant::Billing::CreditCard objects and address hashes in the format accepted by ActiveMerchant in most tests. The functional tests will use the credit card hashes and address hashes which mimic data as it would be posted by a customer. To avoid having to type all of the data to create the objects and hashes in every setup() method of every test class, it is much simpler to add a few simple helpers.

We can use methods called credit_card_hash() and credit_card() to create hashes of credit card data and ActiveMerchant::Billing::CreditCard objects respectively. The credit_card_hash() method returns a hash of credit card data that is updated with the passed in options hash, if one is provided. Add the following to the Test::Unit::TestCase class definition in the test helper:

application/moneyhats/test/test_helper.rb

```
def credit_card_hash(options = {})
  { :number     => '1',
    :first_name => 'Cody',
    :last_name  => 'Fauser',
    :month      => '8',
    :year       => "#{ Time.now.year + 1 }",
    :verification_value => '123',
    :type       => 'visa'
  }.update(options)
end
```

Now add the `credit_card()` method, which uses the `credit_card_hash()` method and returns an `ActiveMerchant::Billing::CreditCard` object:

```
def credit_card(options = {})
  ActiveMerchant::Billing::CreditCard.new( credit_card_
hash(options) )
end
```

By default, the `credit_card()` method returns a credit card which will cause a successful response when used with the Bogus payment gateway. You can simply override the various attributes of the card by passing in an `options` hash if you want to simulate a failing transaction. We'll see how to use this helper shortly.

Addresses are pain to type out in full every time, and the data required rarely changes, so define the following `address()` method in the test helper:

application/moneyhats/test/test_helper.rb

```
def address(options = {})
  { :name     => 'Cody Fauser',
    :address1 => '2500 Oak Mills Road',
    :address2 => 'Suite 1000',
    :city     => 'Beverly Hills',
    :state    => 'CA',
    :country  => 'US',
    :zip      => '90210'
  }.update(options)
end
```

Testing with the Bogus Gateway

In order to facilitate testing, ActiveMerchant provides the `ActiveMerchant::Billing::BogusGateway`. The Bogus gateway is a fake gateway that is perfect for testing and saves you the work of mocking remote interactions. All tests written using the the Bogus gateway will run very quickly because no communication is made to external servers. The Bogus gateway works by returning a pre-defined response based on the credit card number or transaction reference used for the transaction.

Let's add some simple tests that will use the Bogus gateway to test a payment authorization. Open up the file `test/unit/order_transaction_test.rb` and add the following code which tests successful authorization, failed authorization, and an exception being raised during authorization:

application/moneyhats/test/unit/order_transaction_test.rb

```
def setup
  @amount = 100
end

def test_successful_authorization
  auth = OrderTransaction.authorize(
            @amount,
            credit_card(:number => '1')
          )
  assert auth.success
  assert_equal 'authorization', auth.action
  assert_equal BogusGateway::SUCCESS_MESSAGE, auth.message
  assert_equal BogusGateway::AUTHORIZATION, auth[:reference]
end

def test_failed_authorization
  auth = OrderTransaction.authorize(
            @amount,
            credit_card(:number => '2')
          )
  assert !auth.success
  assert_equal 'authorization', auth.action
  assert_equal BogusGateway::FAILURE_MESSAGE, auth.message
end

def test_exception_during_authorization
  auth = OrderTransaction.authorize(
            @amount,
```

AUTHORIZATION CREDIT CARD NUMBERS

When testing **`authorize`**, use the following numbers:

- 1 – Successful Authorization
- 2 – Failed Authorization
- 3 – Raise ActiveMerchant Exception

```
            credit_card(:number => '3')
          )
  assert !auth.success
  assert_equal 'authorization', auth.action
  assert_equal BogusGateway::ERROR_MESSAGE, auth.message
end
```

As you can see from the three simple tests: using the number '1' for the credit card number results in a successful authorization, the number '2' results in a failed authorization, and the number '3' results in an exception being raised during authorization. The tests also ensure that the important data expected in the returned object is present.

Now that we've got some tests for performing authorizations, we can move onto testing the capturing of a payment:

application/moneyhats/test/unit/order_transaction_test.rb

```
def test_successful_capture
  capt = OrderTransaction.capture(@amount, '123')
  assert capt.success
  assert_equal 'capture', capt.action
  assert_equal BogusGateway::SUCCESS_MESSAGE, capt.message
end

def test_failed_capture
  capt = OrderTransaction.capture(@amount, '2')
  assert !capt.success
  assert_equal 'capture', capt.action
  assert_equal BogusGateway::FAILURE_MESSAGE, capt.message
end

def test_error_during_capture
  capt = OrderTransaction.capture(@amount, '1')
  assert !capt.success
  assert_equal 'capture', capt.action
  assert_equal BogusGateway::CAPTURE_ERROR_MESSAGE,
               capt.message
end
```

CAPTURE CREDIT CARD NUMBERS

When testing `capture`, use the following numbers:

- 1 – Failed Capture
- 2 – Raise ActiveMerchant Exception
- Anything else – Success

These tests are similar to the authorization tests except that the test result is based on the authorization reference string passed into the `capture()` call. Once again we're testing a successful response, failure response, and an exception raised during the method call. The three different responses are generated based on the authorization string passed into the `capture()` call. A failure response is generated using the string `'1'`, an exception is raised by using the string `'2'`, and a successful response is returned for any other authorization string.

Using the Bogus gateway throughout your application's unit, functional, and integration tests will ensure the correct behavior of the payment processing code, and keep the tests running quickly. However, testing with the Bogus gateway does not ensure that the application will successfully process payments when connected to the real payment processor. Later, we will write a separate suite of remote tests that interact with the gateway.

Managing Order State

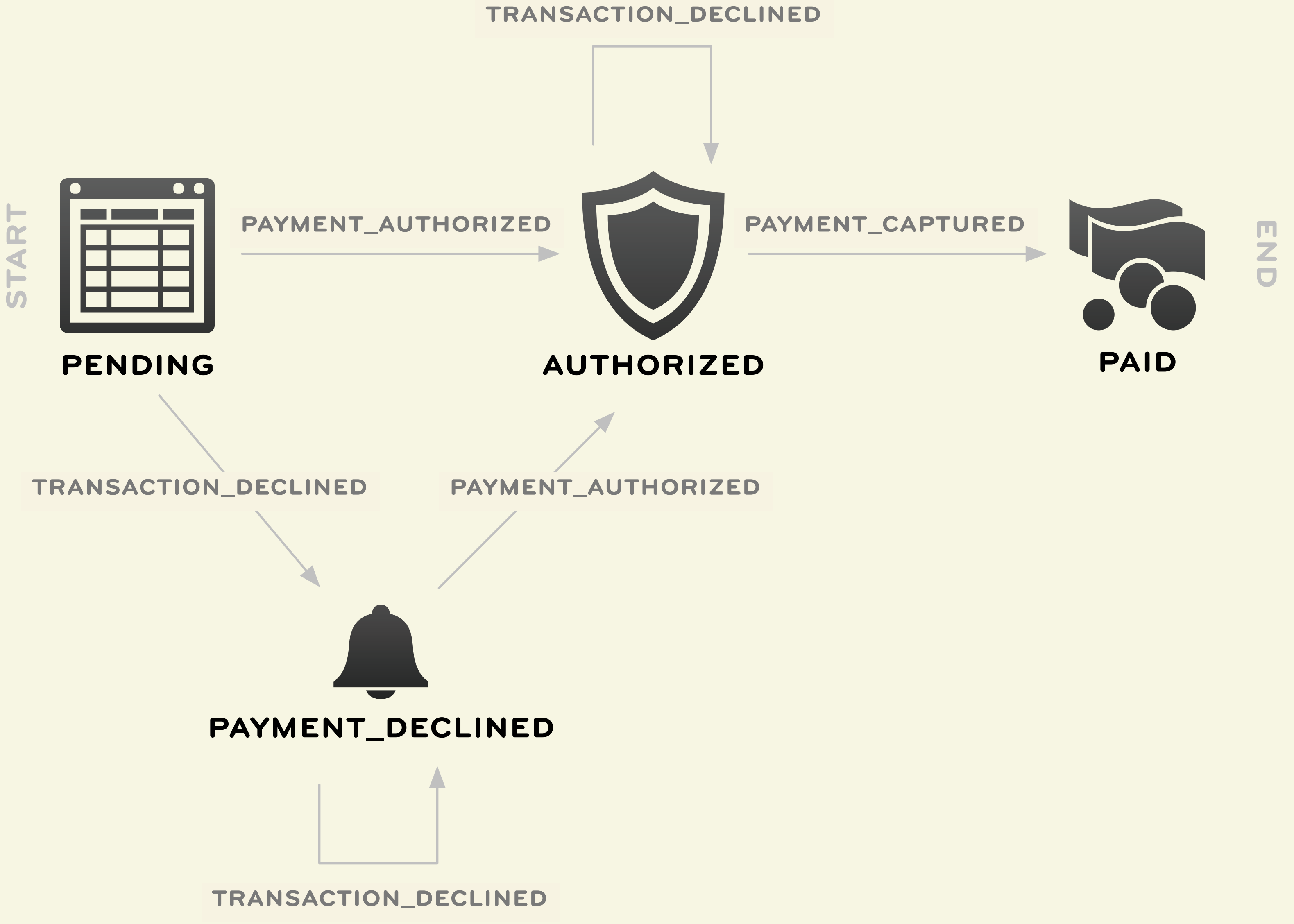
Orders can have several financial states, but only be in one state at any particular time. The state an order is in determines which operations can be made on the order. The management of orders' financial state is important for developing the rest of your application's business logic and UI.

There are many ways we could keep track of the different order states, but for this example we're going to use the excellent Acts as State Machine (AASM) plugin by Scott Barron.

Installation

Installation of the plugin is very simple:

ORDER STATES



```
$ ./script/plugin install http://elitists.textdriven.com/
svn/plugins/acts_as_state_machine/trunk
```

Defining the State Machine

AASM allows us to effortlessly define the possible states orders can have, and the transitions between order states. This will allow the application, particularly the UI, to present the correct actions based on an order's state. For example, it only makes sense to present a button for capturing payment when an order is in the `authorized` state.

The plugin also has many other powerful features, such as callbacks that can be executed before or after entering a particular state. The callbacks are excellent for performing tasks such as sending a notification email when an order enters a particular state.

The state machine could be further expanded to support even more order states such as refunded, partially_refunded, or cancelled, but these additional states are outside the scope of this document.

Now we can easily implement the state machine in the `Order` model:

application/moneyhats/app/models/order.rb

```
acts_as_state_machine :initial => :pending

state :pending
state :authorized
state :paid
state :payment_declined

event :payment_authorized do
  transitions :from => :pending,
              :to   => :authorized

  transitions :from => :payment_declined,
              :to   => :authorized
end

event :payment_captured do
  transitions :from => :authorized,
              :to   => :paid
end

event :transaction_declined do
  transitions :from => :pending,
              :to   => :payment_declined

  transitions :from => :payment_declined,
              :to   => :payment_declined

  transitions :from => :authorized,
              :to   => :authorized
end
```

The initial call to `acts_as_state_machine` enables the state machine on the `Order` model, and sets the initial state of the plugin to `pending`. The default column used to keep track of the model's state is the column `state`, but you can override this by passing the `:column` option with a different column name.

Each `state` declaration defines a valid state for an `Order` object. The states we have defined are as follows:

- `pending` - The initial order state. No successful or unsuccessful payments have been attempted on the order.
- `authorized` - A successful order authorization has been made. The order is ready for fulfillment and payment capture.
- `paid` - The authorized payment has been captured.
- `payment_declined` - Authorization has been attempted, but failed. The order can still be authorized in a subsequent authorization attempt.

Each state declaration also defines a predicate method for the corresponding state in the format `state?`. The defined predicate method indicates whether or not the object is currently in the state in question. We end up with the following methods: `pending?`, `authorized?`, `paid?`, and `payment_declined?`.

Next are the declarations defining the events which will transfer the object between the defined states. The `event` declarations reproduce, in code, the state transitions as they appear in the order state diagram. Each event declaration also defines a method in the format `event!` which causes the state transition and saves the object. From our `event` declarations we get the following methods: `payment_authorized!`, `payment_captured!`, and `transaction_declined!`. The methods are `!` methods to follow the Rails convention that the attribute will be modified and saved in a single step.

> It's not for us to decide whether or not this use of the bang is an appropriate Ruby convention. We encourage you to read David Black's thoughts on the matter (http://dablog.rubypal.com/2007/8/15/bang-methods-or-danger-will-rubyist) and decide for yourself!

Also, notice that the `transaction_declined` event defines the two loopback transitions from `payment_declined` back to itself and from `authorized` back to itself. The definition of the loopback transitions completes the state machine implementation.

With the states and events declared we can move on to writing the code that will perform the authorization and capture of payments. It will definitely be a lot simpler thanks to the Acts as State Machine plugin.

Resources

- Rubyist - Acts as State Machine (http://rubyi.st/2006/1/21/acts-as-state-machine)
- Ruby on Rails Finite State Machine Plugin: acts_as_state_machine (http://rails.aizatto.com/2007/05/24/ruby-on-rails-finite-state-machine-plugin-acts_as_state_machine/)

Authorizing Payment

We're now ready to add some behavior to the `Order` model. The `Order` model is currently empty, aside from the state machine definition. We still need to write the code which authorizes and captures payments. Let's start with the `authorize_payment()`: method:

```
application/moneyhats/app/models/order.rb
def authorize_payment(credit_card, options = {})
  options[:order_id] = number

  transaction do

    authorization = OrderTransaction.authorize(amount,
credit_card, options)
    transactions.push(authorization)

    if authorization.success?
      payment_authorized!
    else
      transaction_declined!
    end

    authorization
  end
end
```

The `authorize_payment()` method takes an `ActiveMerchant::Billing::CreditCard` as its first argument and a hash of options as the second. The `options` hash is passed along to the payment gateway. The `options` hash, as we saw earlier in the document, is how the customer's address and other details about the purchase are passed along to the payment gateway.

The first thing we're doing is setting the `:order_id` in the `options` hash to `Order#number()`. The `:order_id` is an important option for many gateways. Not only does it allow you to more easily correlate transactions between your application and your gateway's admin interface, but some gateways will also require you to send unique order ids with each authorization or purchase. Braintree does not require you to use a unique order ID with each transaction, but they will reject transactions as duplicates if their system sees multiple transactions with the same details too close to one another without a unique order ID. Therefore, it is a good practice to always use a unique id for each new purchase or authorization. `Order#number()` is defined as:

```
def number
  CGI::Session.generate_unique_id
end
```

The order number is being generated randomly using the CGI::Session.generate_unique_id class method. In a real application you would use your organization's order numbering scheme within the application and then stub the `Order#number` method to return the unique identifier for testing purposes. This could easily be done using Rails' built in mocking functionality.

Now the authorization with the payment gateway is performed. First, we create a new `OrderTransaction` object containing the details of the authorization response by calling `OrderTransaction.authorize()` with the `amount`, `credit_card` details, and the `options`. Then we push the `authorization` into the `transactions` collection of the order.

Now we trigger the order's state transition. If the `authorization` was successful we call `payment_authorized!` to put the order into the `authorized` state. If the transaction was not successful then the order is placed into the `payment_declined` state by calling `transaction_declined!`. Finally, the `authorization` is returned by the method. All of this is wrapped in a transaction block to ensure atomicity of the transaction in case disaster strikes.

The `authorization` is saved during the order's state change. This happens automatically because the state transition saves the order, which in turn, saves all unsaved child objects of the order.

What about credit card validation? The credit card is not validated in `authorize_payment()` because we're giving the method a precondition that the card is valid. The card validation would take place in the controller action where the call to `authorize_payment()` is made. There is no reason to execute `authorize_payment()` with an invalid credit card because we know that the authorization cannot ever be successful.

> The Acts as State Machine event methods, such as `payment_authorized!`, can be executed arbitrarily. Currently, there is no mechanism in place to ensure that the appropriate gateway operation has been performed that matches the state transition. This would allow an order to transition to a paid state without any funds actually being transferred to the merchant. Fortunately, Acts as State Machine also takes an optional `:guard` option when declaring each state. The `:guard` option takes either a `Proc` that receives the current object as its argument, or a symbol referencing a method name. The guard prevents the object from transitioning into the next state unless the `Proc` or method executed returns true. The method or `Proc` would be defined to only return `true` if a successful `OrderTransaction` existed for the appropriate action.

There is one last method to define before we write the unit tests for the Order model. The method is called authorization_reference(). The method returns the reference string from the initial successfully authorized transaction. The reference string from the authorization is required for any type of reference transaction, such as capturing payment. The method looks like:

application/moneyhats/app/models/order.rb

```
def authorization_reference
  if authorization = transactions.find_by_action_and_
success('authorization', true, :order => 'id ASC')
    authorization.reference
  end
end
```

Now that the Order model can authorize payment let's move along and write a few unit tests to ensure that the code works the way we expect.

Testing Payment Authorization

First, let's define a few Order fixtures that we'll use in our test cases:

application/moneyhats/test/fixtures/orders.yml

```
pending:
  description: pending order
  amount: 100
  state: pending

authorized:
  description: authorized order
  amount: 100
  state: authorized

uncapturable:
  description: authorized, but uncapturable
  amount: 100
  state: authorized

uncapturable_error:
  description: authorized, but uncapturable due to error
  amount: 100
  state: authorized
```

All of the orders, except the `pending` order, need child order transactions defined:

application/moneyhats/test/fixtures/order_transactions.yml

```
  successful_authorization:
    action: authorization
    order: authorized
    amount: 100
    success: true
    reference: 123
    message: The transaction was successful

  authorization_with_failing_reference:
    action: authorization
    order: uncapturable
    amount: 100
    success: true
    reference: 2
    message: The transaction was successful

  authorization_with_error_reference:
    action: authorization
    order: uncapturable_error
    amount: 100
    success: true
    reference: 1
    message: The transaction was successful
```

WHERE ARE THE FOREIGN KEYS?

These fixtures use a new feature of Rails 2. Instead of specifying a numeric `order_id`, you can use the name of the order fixture you want to reference (`authorized`, `uncapturable`, `uncapturable_error`).

The fixtures' references are the same as we used earlier when unit testing the `OrderTransaction` model.

Let's add the tests that test the basic success, failure, and error responses that the Bogus payment gateway offers us. Open up `test/unit/order_test.rb` and add the following:

application/moneyhats/test/unit/order_test.rb

```
def test_successful_order_authorization
  order = orders(:pending)
  credit_card = credit_card(:number => '1')

  assert_difference 'order.transactions.count' do
    authorization = order.authorize_payment(credit_card)
    assert_equal authorization.reference,
                 order.authorization_reference
    assert authorization.success?
    assert order.authorized?
  end
end

def test_failed_order_authorization
  order = orders(:pending)
  credit_card = credit_card(:number => '2')

  assert_difference 'order.transactions.count' do
    authorization = order.authorize_payment(credit_card)
    assert_nil order.authorization_reference
    assert !authorization.success?
    assert order.payment_declined?
  end
end

def test_exception_raised_during_authorization
  order = orders(:pending)
  credit_card = credit_card(:number => '3')

  assert_difference 'order.transactions.count' do
    authorization = order.authorize_payment(credit_card)
    assert_nil order.authorization_reference
    assert !authorization.success?
    assert order.payment_declined?
  end
end
```

Taking a closer look at the three tests: first, an order in the **pending** state is chosen in each test. Then an authorization is made with a credit card that will return the desired response. Next the test asserts that the authorization_reference is set in the case of a successful response and nil in the case of a failure, or error. The success of the authorization is asserted on the next line and should only be true in the in the case of the successful authorization. Lastly the the tests ensure that the order ends up in the correct state that is expected: authorized in the case of success, payment_declined otherwise. All of these assertions are wrapped in an assert_difference block that ensures that an OrderTransaction object persisted to the database.

Now that the authorize_payment() method is finished, and has a bit of test coverage, let's move on to capturing payment.

Capturing Payment

Authorizing the credit card was only the first half of our implementation of the Authorization & Capture model. Fortunately, implementing capture is very easy because the capture_payment() method is very similar to the authorize_payment() method that we just wrote. Open up the Order model and define it as follows:

application/moneyhats/app/models/order.rb

```
def capture_payment(options = {})
  transaction do
    capture = OrderTransaction.capture(amount, authorization_
reference, options)
    transactions.push(capture)
    if capture.success?
      payment_captured!
    else
      transaction_declined!
    end

    capture
  end
end
```

First, `OrderTransaction.capture()` is called. The first argument is the amount to be captured. For our application the amount is the total order amount `Order#amount`. The `authorization_reference` is the reference string from the initial successful authorization. Any optional information is passed in through the `options` hash. The call to `OrderTransaction.capture()` returns a new `OrderTransaction` that records the details of the capture transaction. The transaction result `capture` is then pushed into the `transactions` collection, just as we did with authorization.

Next, the state of the order is updated. The `capture_successful!` event is called if the capture was successful, or the `transaction_declined!` event is called it wasn't. The `capture_successful!` event will cause the order to transition into the `paid` state, whereas the `transaction_declined!` event is a loopback to the `authorized` state. As we did in `authorize_payment()`, the code is wrapped within a `transaction` block to ensure atomicity. Finally, the `OrderTransaction` object stored in the local variable `capture` is returned from the method.

Let's add some some unit tests for `capture_payment()`.

Testing Payment Capture

You probably already know how the tests we're going to for write `capture_payment()` will look. They are very similar to the tests we already wrote for `authorize_payment()` and build upon the tests that we wrote for `OrderTransaction.capture()`:

application/moneyhats/test/unit/order_test.rb

```ruby
def test_successful_payment_capture
  order = orders(:authorized)

  assert_difference 'order.transactions.count' do
    capture = order.capture_payment
    assert order.paid?
    assert capture.success?
  end
end

def test_failed_payment_capture
  order = orders(:uncapturable)

  assert_difference 'order.transactions.count' do
    capture = order.capture_payment
    assert order.authorized?
    assert !capture.success?
  end
end

def test_error_during_payment_capture
  order = orders(:uncapturable_error)

  assert_difference 'order.transactions.count' do
    capture = order.capture_payment
    assert order.authorized?
    assert !capture.success?
  end
end
```

The tests are using the order fixtures `authorized`, `uncapturable`, and `uncapturable_error` respectively. The fixtures are orders containing authorization transactions with reference strings that will cause the desired outcome we want for each test. The first test successful captures the authorized payment, asserts that the `OrderTransaction` was successful and asserts that order transitions to the `paid` state. The next two tests test the failure and error responses respectively. Both of the tests assert that the `OrderTransaction` returned is unsuccessful and that the order stays in the `authorized` state. Finally, all of the tests use `assert_difference` to ensure that an `OrderTransaction` object is saved to the database with the transaction details.

With these unit tests completed we will now write some more remote tests to ensure that both `authorize_payment()` and `capture_payment()` work correctly using the gateway's test servers.

Integration Test Suite

In order to ensure that our application will successfully process payments when connected to our actual payment provider's systems we need a suite of integration tests.

> The term "integration test" in this section refers to tests that interact with third party services. Rails also uses the term "integration test" in a different context.

One method for ensuring that our application is integrated correctly to the payment processor would be to completely ignore the Bogus gateway and use the real payment gateway in test mode, and with test credentials, for all of the tests. This would certainly ensure that our code is integrated correctly. Unfortunately, this would hammer the gateway provider's test servers, and we'd also end up with slow tests.

The simplest solution is to create a special test suite that tests the integration of the application code with the payment gateway's test environment. The tests will not be executed on every invocation of `rake test`. The tests could be run manually, or be executed by a continuous integration server after every code check-in.

Creating the Remote Test Suite

Thankfully with Rails there is an easy way to have a suite of integration tests alongside the normal Rails unit, functional, and integration tests. We're going to create an additional set of tests that will be invoked by `rake test:remote`. You can use this new set of tests to test all of your interactions with 3rd party services, not just for ActiveMerchant gateways.

First, we'll create a new folder where the tests will reside, including sub-folders for each type of Rails tests: unit, functional, and integration:

```
mkdir test/remote
mkdir test/remote/unit
mkdir test/remote/functional
mkdir test/remote/integration
```

Now we'll create two new Rake tasks. The first will execute all of the remote tests when we execute `rake test:remote` and the second will execute all tests including remote tests. Create a new file `lib/tasks/remote.rake` and place the following code, modeled from the standard Rails test tasks, in it:

application/moneyhats/lib/tasks/remote.rake

```
namespace :test do

  Rake::TestTask.new(:remote => "db:test:prepare") do |t|
    t.libs << "test"
    t.pattern = 'test/remote/**/*_test.rb'
    t.verbose = true
  end
  Rake::Task['test:remote'].comment = "Test integration with
remote services"

  desc 'Test all unit, functional, integration, and remote
tests'
  task :all do
    errors = %w(test:units test:functionals test:integration
test:remote).collect do |task|
      begin
        Rake::Task[task].invoke
        nil
      rescue => e
        task
      end
    end.compact
    abort "Errors running #{errors.to_sentence}!" if errors.
any?
  end
```

```
end
```

The new task simply runs all the tests in all of the files ending in `_test.rb` in the `test/remote` folder we just created. The task depends on the `db:test:prepare` task, which prepares the test database and loads the schema. Notice that there is also a second task called `test:all` that will execute all tests including the remote tests that we're adding. This task can be executed anytime you want to test your entire application including your integration with any remote services.

Adding Remote Tests

Now that there is a place for the remote integration tests, let's add a few of them. We're going to write tests that verify that the `Order` model functions correctly when the `OrderTransaction` model is configured to send requests to the payment gateway's test environment. Obviously, the tests presented aren't exhaustive. There many additional cases that you'd want to cover in your real application.

Let's write tests that test following basic scenarios: successful authorization, unsuccessful authorization, successful authorization and capture, and unsuccessful capture. Create the file `test/remote/unit/order_test.rb` and add the following code:

application/moneyhats/test/remote/unit/order_test.rb

```
require File.dirname(__FILE__) + '/../../test_helper'

class OrderIntegrationTest < Test::Unit::TestCase

  def setup
    OrderTransaction.gateway = BraintreeGateway.new(
                                 :login    => 'demo',
                                 :password => 'password'
                               )
    @order = orders(:pending)
    @credit_card = credit_card(:number => '4111111111111111')
    @options = { :description => 'A store purchase',
                 :billing_address => address
               }
  end

  def test_successful_order_authorization
    assert_difference '@order.transactions.count' do
      authorization = @order.authorize_payment(@credit_card,
@options)
      assert_equal authorization.reference,
                   @order.authorization_reference
      assert authorization.success?
      assert @order.authorized?
    end
  end

  def test_authorization_unsuccessful_with_invalid_credit_
card_number
    @credit_card.number = 'invalid'

    assert_difference '@order.transactions.count' do
      authorization = @order.authorize_payment(@credit_card,
@options)
      assert_nil @order.authorization_reference
      assert !authorization.success?
      assert @order.payment_declined?
    end
  end

  def test_successful_authorization_and_capture
    assert_difference '@order.transactions.count', 2 do
      authorization = @order.authorize_payment(@credit_card,
@options)
      assert authorization.success?
      assert @order.authorized?

      capture = @order.capture_payment
      assert capture.success?
      assert @order.paid?
    end
  end
  def test_authorization_and_unsuccessful_capture
    assert_difference '@order.transactions.count', 2 do
      authorization = @order.authorize_payment(@credit_card,
@options)
      assert authorization.success?
      assert @order.authorized?

      authorization_transaction = @order.transactions.first
      authorization_transaction.update_attribute(:reference,
                                                 '')

      capture = @order.capture_payment
      assert !capture.success?
      assert @order.authorized?
    end
  end
end
```

The `setup()` method constructs the payment gateway, and sets the `OrderTransaction.gateway` accessor. Then the `@order` instance variable is set to the `pending` order fixture for convenience. Then a new credit card is constructed with a valid number for the test environment and assigned to the instance variable `@credit_card`. Finally, we create a hash of optional data including the customer's billing address and a short description for the purchase. The hash of options is stored in the `@options` instance variable.

The tests are all straight forward and look similar to the unit tests we wrote for the order model. One major difference is that we can't use the fixtures to the same extent we did in the normal unit tests. This is because the transaction reference strings returned by the gateway will always be changing. This also means that we have to test multiple step operations that depend on one another, such as authorization and capture, in a single test.

The first test, `test_successful_order_authorization`, tests successful authorization and is basically the same as the unit test we wrote earlier.

The second test, `test_authorization_unsuccessful_with_invalid_credit_card_number` tests unsuccessful authorization of a payment. We are passing in an invalid credit card number to ensure that the transaction will not be successful.

Then `test_successful_authorization_and_capture` tests successful authorization and capture of an order. The order transitions from the `pending` state to the `authorized` state and then ends up in the `paid` state after being successful captured. As mentioned above, both operations must take place in the single test because the payment capture must reference the transaction reference from the authorization.

Finally, `test_authorization_and_unsuccessful_capture` tests a failing payment capture by attempting to capture an authorization with an invalid transaction `reference`. We do this by updating the `OrderTransaction` with an invalid `reference` string. The gateway will be unable to locate the initial transaction from the invalid `reference` string provided. This will cause the transaction to be declined.

The entire remote test suite can be run with the rake command `rake test:remote`. Running the test suite on my machine gives the following result:

```
cody:moneyhats cody$ rake test:remote
(in /Users/cody/code/peepcode/activemerchant/project/code/
application/moneyhats)
/System/Library/Frameworks/Ruby.framework/Versions/1.8/usr/
bin/ruby -Ilib:test "/Library/Ruby/Gems/1.8/gems/rake-0.8.0/
lib/rake/rake_test_loader.rb" "test/remote/unit/order_test.
rb"
Loaded suite /Library/Ruby/Gems/1.8/gems/rake-0.8.0/lib/
rake/rake_test_loader
Started
....
Finished in 11.273599 seconds.
```

```
  4 tests, 18 assertions, 0 failures, 0 errors
```

All the tests pass! One thing to note is the amount of time taken to run the tests. The 4 tests took 11.273599 seconds. Running the entire unit test suite with the Bogus gateway took only 0.11748 seconds on my machine. Development would slow to a crawl without separation of the remote tests to their own test suite.

The tests we just wrote are testing the integration to the payment gateway at the unit test level. These tests are important and ensure that the `Order` model works as expected. It could also be worthwhile to add functional and integration tests to the remote test suite. Functional and integration tests test your application at a deeper level and can help you ensure that your application functions correctly from end to end when connected to the payment gateway.

Summary

We successfully developed a simple application that can authorize and capture funds from credit cards. This is only a sliver of a real application, but it should provide the knowledge necessary to get you started with adding credit card processing to your own application.

There are many parts of the sample application which could be expanded upon. The state machine diagram for the `Order` model was deliberately limited to authorization and capture to keep the application simple. It would be quite easy to expand the `Order` model we developed to support more payment gateway operations such as void and credit. The `OrderTransaction` model was developed in such a way that each additional operation can be added with 3 lines of code.

Another piece of the application that could be expanded upon is the communication of additional transaction information to the payment gateway. We only sent the gateway the minimum amount of information to produce successful transactions. Sending the customer's billing address, shipping address, email, or other data to the payment gateway would be as easy as adding the data to the `options` hash sent to `OrderTransaction.authorize()` and `OrderTransaction.capture()`.

Payment processing isn't all fun and games. In the next section we're going to take a look at some of the security considerations you should keep in mind when developing your application.

Security

CHAPTER 4

This section talks about security and the Payment Card Industry (PCI) Data Security Standard (DSS). This section discusses secure communication of your application over public networks, filtering of cardholder information from log files and stack traces, storage of cardholder information, and relates the topics to the PCI DSS. This section only deals with your Rails application code, not with server security.

PCI Data Security Standard

The Payment Card Industry (PCI) Data Security Standard (DSS), originally created by Visa and MasterCard, is a set of comprehensive requirements for enhancing payment account data security. The PCI DSS is important to you as an application developer because any company that accepts, processes, or stores credit card information needs to comply with the standards set by the Payment Card Industry.

All of the major credit card companies use the PCI DSS as the basis for their cardholder information security programs. Merchants and service providers must meet the security requirements of each credit card brand that they process. Failing to comply with a card company's security programs can result in fines, or even suspension of the ability to process the company's card type.

The PCI DSS includes requirements for all of the following:

- security management
- policies

- procedures
- network architecture
- software design and other critical protective measures

This section relates to the Protect Cardholder Data section of the the PCI DSS v1.1, which includes Requirement 3: Protect stored cardholder data, and Requirement 4: Encrypt transmission of cardholder data across open, public networks. This section is by no means exhaustive, or authoritative, but merely intends to provide some tips that will hopefully put your Rails application a few steps closer to PCI compliance.

Resources

- Payment Card Industry Security Standards Council (https://www.pcisecuritystandards.org/index.htm)
- Payment Card Industry Data Security Standard v1.1 (https://www.pcisecuritystandards.org/pdfs/pci_dss_v1-1.pdf)
- Visa USA Cardholder Information Security Program (CISP) (http://usa.visa.com/merchants/risk_management/cisp.html)
- MasterCard International Site Data Protection (SDP) Program (https://www.mastercard.com/us/sdp/index.html)
- American Express Data Security Operating Policy (DSOP) Program (http://www125.americanexpress.com/merchant/oam/ns/USEng/FrontServlet?request_type=navigate&page=dataSecurityRequirements)
- Discover Card Information Security and Compliance (DISC) Program (http://www.discovernetwork.com/merchant/resources/data/data_security.html)
- JCB PCI DSS (http://www.jcb-global.com/english/pci/index.html)

SSL

Requirement 4 of the PCI DSS v1.1 stipulates that all cardholder data transmitted across open, public networks must be encrypted. For a Rails application this means that you have to ensure that all communications with controller actions which send or receive cardholder data are made using an SSL connection.

Cardholder information is also transmitted and received by ActiveMerchant when communicating with payment gateways. ActiveMerchant ensures that all connections to payment gateways are made using SSL. ActiveMerchant also verifies the server's SSL certificate with the signing certificate of the major certificate authorities to prevent spoofing.

The easiest way to ensure that your application uses SSL for sensitive controller actions is to install the SSL Requirement plugin. Install the plugin:

```
./script/plugin install ssl_requirement
```

Now that the plugin has been installed it needs to be required into the `ApplicationController` of your application in `app/controllers/application.rb`:

```
class ApplicationController < ActionController::Base
  include SslRequirement
end
```

The example from the README shows how to use the plugin:

```
examples/ssl_requirement.rb
class AccountController < ApplicationController
  ssl_required :signup, :payment
  ssl_allowed  :index
```

```ruby
  def signup
    # Non-SSL access will be redirected to SSL
  end

  def payment
    # Non-SSL access will be redirected to SSL
  end

  def index
    # This action will work either with or without SSL
  end

  def other
    # SSL access will be redirected to non-SSL
  end
end
```

Take a look at the full README file from the plugin for more information.

The Development Environment

The plugin works well out of the box, but there isn't much point in requiring SSL in the development environment. In order to disable the functionality of the plugin in the development environment you'll want to stub out the plugin's `ssl_required?` method. This can be done using Rails' built in mocking facilities. Add the file `test/mocks/development/application.rb` and add the following code to it:

examples/ssl_requirement.rb

```ruby
require 'controllers/application'

class ApplicationController < ActionController::Base
  def ssl_required?
    false
  end
end
```

The code requires the real `ApplicationController` and then stubs out the `ssl_required?` method to ensure that SSL is never required in the development environment.

The Test Environment

For functional and integration tests you'll also need to indicate that the request is using SSL for actions which require SSL. This is can be done by setting the HTTPS environment variable of the request to `on`, like in the following functional test example:

examples/ssl_requirement.rb

```ruby
class AccountControllerTest < ActionController::TestCase
  def test_get_signup
    get :signup
    assert_response :redirect
  end

  def test_get_signup_with_ssl
    @request.env["HTTPS"] = "on"
    get :signup
    assert_response :success
  end
end
```

The SSL Requirement plugin is a quick way to ensure that your application is requiring SSL in your application. The code for the plugin is very simple, so don't hesitate to take a look under the hood, or do some customization if you need to.

Rails Logging

Requirement 3 of the PCI DSS v1.1 is titled Protect Stored Cardholder

Data. If you don't have your Rails logging configured correctly, you could be inadvertently storing cardholder data (including the Card Verification Value) in your application's log files.

Rails does extensive logging of requests including the parameters of the request. For example, the following is a GET request sent to the `index` action of a controller named `CheckoutController`:

```
Processing CheckoutController#index (for 127.0.0.1 at
2007-09-30 20:41:02) [GET]
  Parameters: {"action"=>"index", "controller"=>"checkout"}
```

The verbosity and clarity of the Rails log files makes them very useful, but log files are definitely not a place you want to accumulate cardholder information. Take a look at a POST to a controller where a customer has entered credit card information:

```
Processing CheckoutController#payment (for 127.0.0.1 at
2007-12-03 00:41:34) [POST]
  Parameters: {"commit"=>"Submit", "credit_
card"=>{"month"=>"10", "number"=>"4433221111223344",
"type"=>"visa", "verification_value"=>"222", "year"=>"2009",
"first_name"=>"Cody", "last_name"=>"Fauser"},
"action"=>"payment", "id"=>"1", "controller"=>"checkout",
"billing_address"=>{"name"=>"Cody Fauser", "address1"=>"66
Oak Boulevard", "city"=>"Beverly Hills", "address2"=>"Suite
222", "zip"=>"90210", "country"=>"US", "state"=>"CA"}}
```

Thankfully, Rails provides an easy solution to this. The solution is to use the `filter_parameter_logging` declaration in your controller. For maximum convenience and security, put the following declaration in the `ApplicationController`:

```
filter_parameter_logging :credit_card
```

Placing the declaration in the `ApplicationController` means that you won't have to worry about making the declaration in each controllers where you might be accepting credit cards.

Now let's take another look at the same post we saw above, but with the `filter_parameter_logging` of the credit card information enabled:

```
Processing CheckoutController#payment (for 127.0.0.1 at
2007-12-03 00:33:34) [POST]
  Parameters: {"commit"=>"Submit",
"action"=>"payment", "credit_card"=>"[FILTERED]",
"id"=>"1", "controller"=>"checkout", "billing_
address"=>{"city"=>"Beverly Hills", "address1"=>"66
Oak Boulevard", "name"=>"Cody Fauser", "zip"=>"90210",
"address2"=>"Suite 222", "country"=>"US", "state"=>"CA"}}
```

As you can see, the credit card information has been removed from the log file by Rails (along with all the sub-values of the `credit_card` key in the parameters). Parameter log filtering also helps keep exceptions logs free from credit card information, as we'll see next.

Exception Logging

When an unhandled exception occurs in a Rails application, it makes sense to notify the development team of the problem. However, this is another area of a Rails application that could end up inadvertently storing cardholder information is exception logging or exception notifications. Even worse, email is unencrypted!

The danger of storing or sending notifications of an exception's details lies not with the exception itself, but with the additional contextual information that is required to make the report useful. This additional information includes all of the pertinent details from the current request at the time of the exception. This usually includes `ActionController::AbstractRequest#parameters` and `ActionContro` ller::AbstractRequest#env which both return hashes. The parameters hash contains all of the GET and POST parameters from the request. The env hash contains a hash of environment variables for the request.

Any sensitive data in the parameters or env hashes needs to be filtered manually before storing or reporting the exception. Cardholder information, including the credit card number and CVV2 code, can appear in the request parameters and in the RAW_POST_DATA of the request environment.

If you don't want to roll your own solution, there is the Exception Notifier Plugin (http://dev.rubyonrails.org/svn/rails/plugins/exception_notification/README) by Jamis Buck (http://weblog.jamisbuck.org/). The plugin emails you a summary of any unhandled exceptions that occur in your application. The Exception Notifier plugin automatically filters sensitive information from the exception notifications based on your filter_parameter_logging declarations. This means that the plugin can be used without any risk of cardholder data appearing in the notification, as long as you have made the correct filter_parameter_logging declaration in your controllers.

> Another existing plugin that you may want to consider is Rick Olson's Exception Logger Plugin (http://svn.techno-weenie.net/projects/plugins/exception_logger/). This is based on the Exception Notifier but logs errors to the database instead of sending them out over email.

Let's create a tiny sample application, ExceptionFiltering, with a single class named ExceptionReport which will log exceptions to the database safely. The code is based on the code the Exception Notifier plugin uses for filtering sensitive data from exception notifications. Let's get started:

```
$ rails exception_filtering
$ cd exception_filtering
$ ./script/generate model exception_report params:text
env:text
$ rake db:migrate
```

The model `ExceptionReport` will log exception data, and the relevant request information to the database. The model would be used in your `ApplicationController` or other controller as follows:

examples/snippets.rb

```ruby
def rescue_action_in_public(exception)
  ExceptionReport.create(
    :exception  => exception,
    :controller => self
  )
end
```

You'll probably have a lot more logic in your `rescue_action_in_public()` method, but this is enough for illustrative purposes. Let's write the model code. Open up `app/models/exception_report.rb` and add the following code:

application/exception_filtering/app/models/exception_report.rb

```ruby
class ExceptionReport < ActiveRecord::Base
  PARAM_FILTER_REPLACEMENT = "[FILTERED]"

  serialize :params
  serialize :env

  attr_accessor :exception, :controller

  def filter_parameters(params)
    if controller.respond_to?(:filter_parameters)
      controller.filter_parameters(params)
    else
      controller.params
    end
  end
```

```
    def filter_environment(env)
      return env unless controller.respond_to?(:filter_
  parameters)

      env.inject({}) do |hash, (k, v)|
        if (k =~ /RAW_POST_DATA/i)
          hash[k] = PARAM_FILTER_REPLACEMENT
        else
          hash[k] = controller.filter_parameters({
            k => v
          }).values[0]
        end
        hash
      end
    end

    protected
    def before_create
      self.params = filter_parameters(controller.request.
  parameters)
      self.env    = filter_environment(controller.request.env)
    end
  end
```

The model has two serialized columns, `params` and `env`, for storing the controller context the exception was generated in. We're going to completely ignore the data contained in the exception.

All of the work is done in the `filter_parameters()` and `filter_environment()` instance methods. The `filter_parameters()` method reuses `ActionController::Base#filter_parameters()`, if it has been defined, which takes a hash of parameters and returns a new hash with filtered values replaced by `[FILTERED]`. Otherwise the method returns the hash unaltered.

The `filter_environment()` method first checks to see if `filter_parameters()` has been defined in the controller. If it hasn't then there isn't any sensitive data to filter. If the method has been defined then the method proceeds to filter the environment information. The code iterates through each key value pair in the `env` hash. The method will completely replace the entire `RAW_POST_DATA` with FILTERED and pass all other key value pairs through the `filter_parameters()` method of the controller. The method then returns the new hash.

The model then uses the `before_create` callback to pull the `params` and `env` data from the controller, filter it, and store the filtered data. The entire model is very simple, thanks to the Rails support for filtering parameter logging. Take a look at `test/unit/test_exception_report.rb`, which is provided with the sample code for this document, to see a few tests for the `ExceptionReport` model.

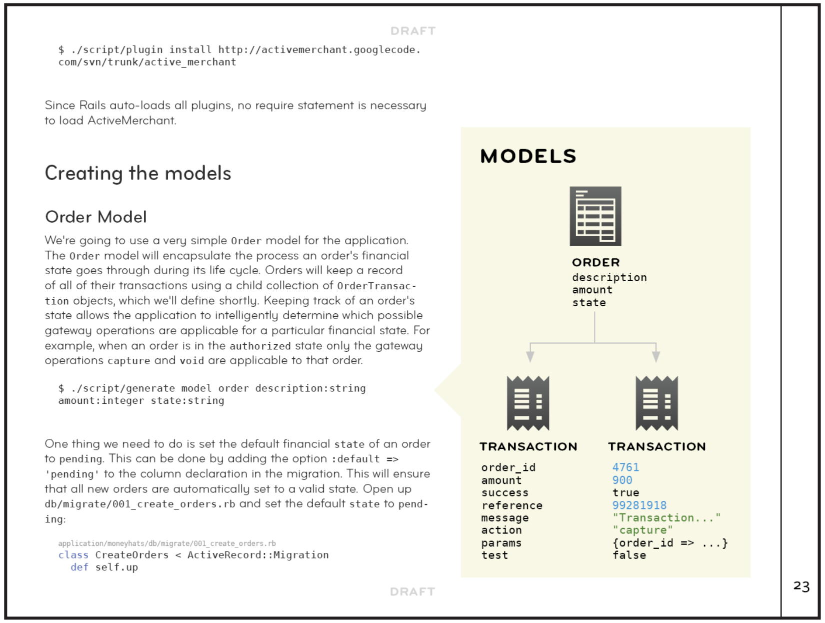
The code presented above is incredibly simple, but provides the tools to help you filter sensitive information from the exception logging of your application.

How to Read a PeepCode Book

PeepCode books are formatted for the screen and look best in full screen mode.

ADOBE READER

Choose View → Full Screen Mode. You can click on the table of contents to jump to a page, or advance with the spacebar, page down, the right arrow key, or the down arrow key.



```
$ ./script/plugin install http://activemerchant.googlecode.
com/svn/trunk/active_merchant
```

Since Rails auto-loads all plugins, no require statement is necessary to load ActiveMerchant.

Creating the models

Order Model

We're going to use a very simple `Order` model for the application. The `Order` model will encapsulate the process an order's financial state goes through during its life cycle. Orders will keep a record of all of their transactions using a child collection of `OrderTransaction` objects, which we'll define shortly. Keeping track of an order's state allows the application to intelligently determine which possible gateway operations are applicable for a particular financial state. For example, when an order is in the `authorized` state only the gateway operations `capture` and `void` are applicable to that order.

```
$ ./script/generate model order description:string
amount:integer state:string
```

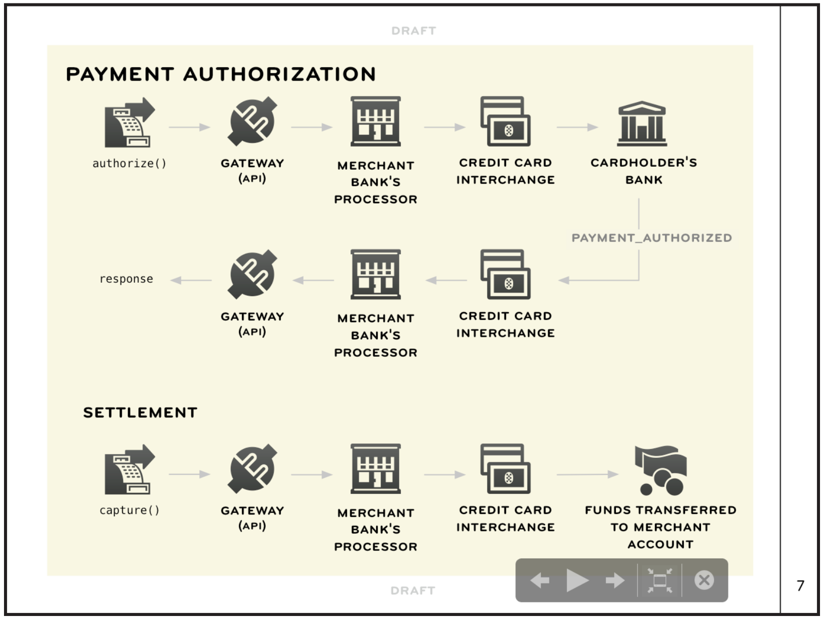
One thing we need to do is set the default financial `state` of an order to `pending`. This can be done by adding the option `:default => 'pending'` to the column declaration in the migration. This will ensure that all new orders are automatically set to a valid state. Open up `db/migrate/001_create_orders.rb` and set the default `state` to `pending`:

```
application/moneyhats/db/migrate/001_create_orders.rb
class CreateOrders < ActiveRecord::Migration
  def self.up
```

TRANSACTION	TRANSACTION
order_id	4761
amount	900
success	true
reference	99281918
message	"Transaction..."
action	"capture"
params	{order_id => ...}
test	false

APPLE PREVIEW

Choose View → Slideshow, then click the Fit to Screen button on the floating control bar. You can advance with the right arrow key.



$9

PeepCode press

Ruby on Rails

Code Review

Improve the quality of your code

by Geoffrey Grosenbach

RAILS CODE REVIEW

by Geoffrey Grosenbach

Seventeen useful tips for improving the design and implementation of your Ruby on Rails application.

Make your application more scalable, reduce duplicated code, and get straight to the point with this useful book.

PeepCode press

$9

Rails2

New features for your applications

by Ryan Daigle

y! Factorial

RAILS 2

by Ryan Daigle

Rails 2.0 has many new features that will save time and make your applications more maintainable. Get the scoop on the best new features from Ryan Daigle, the recognized expert on the latest day-to-day enhancements to Rails.

PeepCode: ActiveMerchant PDF

Every effort was made to provide accurate information in this document. However, neither Cody Fauser nor Topfunky Corporation shall have any liability for any errors in the code or descriptions presented in this book.

This document is available for single users for US$9 at PeepCode.com (http://peepcode.com). Group discounts and site licenses can also be purchased by sending email to peepcode@topfunky.com.

OTHER PEEPCODE PRODUCTS

- RSpec (http://peepcode.com/products/rspec-basics) – A three part series on the popular behavior-driven development framework.
- Rails from Scratch (http://peepcode.com) – Learn Rails!
- RESTful Rails (http://peepcode.com/products/restful-rails) – Teaches the concepts of application design with REST.
- Subscription pack of 10 (http://peepcode.com/products/subscription-pack-of-10) – Save money! Buy 10 PeepCode credits.
- Javascript with Prototype (http://peepcode.com/products/javascript-with-prototypejs) – Code confidently with Javascript!
- Rails Code Review PDF (http://peepcode.com/products/draft-rails-code-review-pdf) – Common mistakes in Rails applications, and how to fix them.