spreecommerce™ BEHIND THE BEST STOREFRONTS™

- Home
- User
- Developer
- Hub
- API
- Release Notes

Payments - Developer Guide | Spree Commerce

Overview

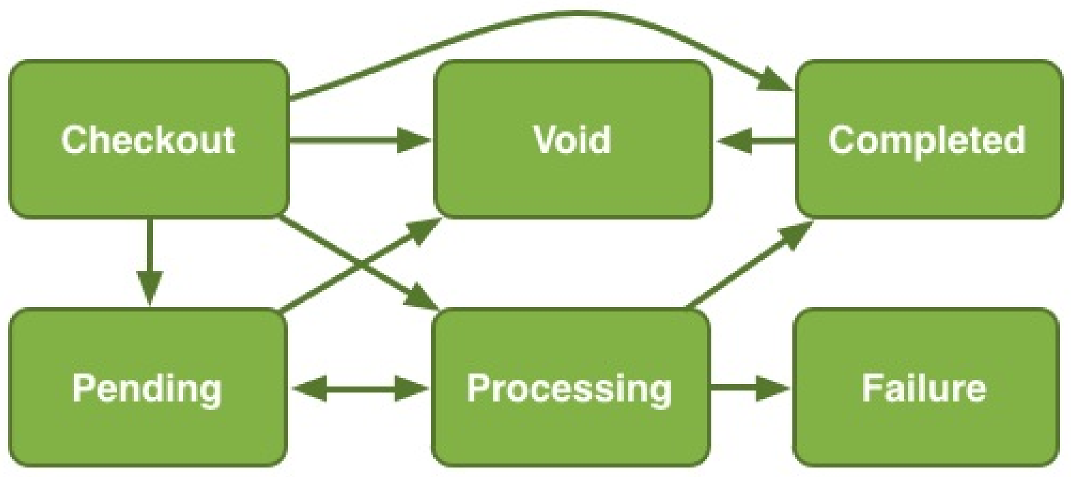
Spree has a highly flexible payments model which allows multiple payment methods to be available during checkout. The logic for processing payments is decoupled from orders, making it easy to define custom payment methods with their own processing logic.

Payment methods typically represent a payment gateway. Gateways will process card payments, and may also include non-gateway methods of payment such as Check, which is provided in Spree by default.

The `Payment` model in Spree tracks payments against Orders. Payments relate to a `source` which indicates how the payment was made, and a `PaymentMethod`, indicating the processor used for this payment.

When a payment is created, it is given a unique, 8-character identifier. This is used when sending the payment details to the payment processor. Without this identifier, some payment gateways mistakenly reported duplicate payments.

A payment can go through many different states, as illustrated below.



An explanation of the different states:

- `checkout`: Checkout has not been completed
- `processing`: The payment is being processed (temporary – intended to prevent double submission)
- `pending`: The payment has been processed but is not yet complete (ex. authorized but not captured)
- `failed`: The payment was rejected (ex. credit card was declined)
- `void`: The payment should not be counted against the order
- `completed`: The payment is completed. Only payments in this state count against the order total

The state transition for these is handled by the processing code within Spree; however, you are able to call the event methods yourself to reach these states. The event methods are:

- `started_processing`
- `failure`
- `pend`
- `complete`
- `void`

Payment Methods

Payment methods represent the different options a customer has for making a payment. Most sites will accept credit card payments through a payment gateway, but there are other options. Spree also comes with built-in support for a Check payment, which can be used to represent any offline payment. There are also third-party extensions that provide support for some other interesting options such as better_spree_paypal_express.

A `PaymentMethod` can have the following attributes:

- `type`: The subclass of `Spree::PaymentMethod` this payment method represents. Uses rails single table inheritance feature.
- `name`: The visible name for this payment method
- `description`: The description for this payment method
- `active`: Whether or not this payment method is active. Set it `false` to hide it in frontend.
- `environment`: The Rails environment (`Rails.env`) where this payment method is active
- `display_on`: Determines where the payment method can be visible. Values can be `front` for frontend, `back` for backend or `both` for both.

Payment Method Visibility

The appearance of the payment methods on the frontend and backend depend on several criteria used by the `PaymentMethod.available` method. The code is this:

```
def self.available(display_on = 'both')
  all.select do |p|
    p.active &&
    (p.display_on == display_on.to_s || p.display_on.blank?) &&
    (p.environment == Rails.env || p.environment.blank?)
  end
end
```

If a payment method meets these criteria, then it will be available.

Auto-Capturing

By default, a payment method's `auto_capture?` method depends on the `Spree::Config[:auto_capture]` preference. If you have set this preference to `true`, but don't want a payment method to be auto-capturable like other payment methods in your system, you can override the `auto_capture?` method in your `PaymentMethod` subclass:

```
class FancyPaymentMethod < Spree::PaymentMethod
  def auto_capture?
    false
  end
end
```

The result of this method determines if a payment will be automatically captured (true) or only authorized (false) during the processing of the payment.

Payment Processing

Payment processing in Spree supports many different gateways, but also attempts to comply with the API provided by the active_merchant gem where possible.

Gateway Options

For every gateway action, a list of gateway options are passed through.

- `email` and `customer`: The email address related to the order
- `ip`: The last IP address for the order
- `order_id`: The Order's `number` attribute, plus the `identifier` for each payment, generated when the payment is first created
- `shipping`: The total shipping cost for the order, in cents
- `tax`: The total tax cost for the order, in cents
- `subtotal`: The item total for the order, in cents
- `currency`: The 3-character currency code for the order
- `discount`: The promotional discount applied to the order
- `billing_address`: A hash containing billing address information
- `shipping_address`: A hash containing shipping address information

The billing address and shipping address data is as follows:

- `name`: The combined `first_name` and `last_name` from the address
- `address1`: The first line of the address information
- `address2`: The second line of address information
- `city`: The city of the address
- `state`: An abbreviated version of the state name or, failing that, the state name itself, from the related

`State` object. If that fails, the `state_name` attribute from the address.

- `country`: The ISO name for the country. For example, United States of America is “US”, Australia is “AU”.
- `phone`: The phone number associated with the address

Credit Card Data

Spree stores only the type, expiration date, name and last four digits for the card on your server. This data can then be used to present to the user so that they can verify that the correct card is being used. All credit card data sent through forms is sent through immediately to the gateways, and is not stored for any period of time.

Processing Walkthrough

When an order is completed in spree, each `Payment` object associated with the order has the `process!` method called on it (unless `payment_required?` for the order returns `false`), in order to attempt to automatically fulfill the payment required for the order. If the payment method requires a source, and the payment has a source associated with it, then Spree will attempt to process the payment. Otherwise, the payment will need to be processed manually.

If the `PaymentMethod` object is configured to auto-capture payments, then the `Payment#purchase!` method will be called, which will call `PaymentMethod#purchase` like this:

```
payment_method.purchase(<amount>, <source>, <gateway options>)
```

If the payment is *not* configured to auto-capture payments, the `Payment#authorize!` method will be called, with the same arguments as the `purchase` method above:

```
payment_method.authorize(<amount>, <source>, <gateway options>)
```

How the payment is actually put through depends on the `PaymentMethod` sub-class’ implementation of the `purchase` and `authorize` methods.

The returned object from both the `purchase` and `authorize` methods on the payment method objects must be an `ActiveMerchant::Billing::Response` object. This response object is then stored (in YAML) in the `spree_log_entries` table. Log entries can be retrieved with a call to the `log_entries` association on any `Payment` object.

If the `purchase!` route is taken and is successful, the payment is marked as `completed`. If it fails, it is marked as `failed`. If the `authorize` method is successful, the payment is transitioned to the `pending` state so that it can be manually captured later by calling the `capture!` method. If it is unsuccessful, it is also transitioned to the `failed` state.

Once a payment has been saved, it also updates the order. This may trigger the `payment_state` to change, which would reflect the current payment state of the order. The possible states are:

- `balance_due`: Indicates that payment is required for this order

- `failed`: Indicates that the last payment for the order failed
- `credit_owed`: This order has been paid for in excess of its total
- `paid`: This order has been paid for in full.

You may want to keep tabs on the number of orders with a `payment_state` of `failed`. A sudden increase in the number of such orders could indicate a problem with your credit card gateway and most likely indicates a serious problem affecting customer satisfaction. You should check the latest `log_entries` for the most recent payments in the store if this is happening.

Log Entries

Responses from payment gateways within Spree are typically `ActiveMerchant::Billing::Response` objects. When Spree handles a response from a payment gateway, it will serialize the object as YAML and store it in the database as a log entry for a payment. These responses can be useful for debugging why a payment has failed.

You can get a list of these log entries by calling the `log_entries` on any `Spree::Payment` object. To get the `Active::Merchant::Billing::Response` out of these `Spree::LogEntry` objects, call the `details` method.

Supported Gateways

Access to a number of payment gateways is handled with the usage of the spree_gateway extension. This extension currently supports the following gateways:

- Authorize.Net
- Balanced
- Beanstram
- Braintree
- eWAY
- LinkPoint
- Moneris
- PayPal
- Sage Pay
- Samurai
- Skrill
- Stripe
- USA ePay
- WorldPay

With the `spree_gateway` gem included in your application's `Gemfile`, these gateways will be selectable in the admin backend for payment methods.

These are just some of the gateways which are supported by the Active Merchant gem. You can see a list of all the Active Merchant gateways on that project's GitHub page.

In order to implement a new gateway in the spree_gateway project, please refer to the other gateways within `app/models/spree/gateway` inside that project.

Adding your custom gateway

In order to make your custom gateway show up on backend list of available payment methods you need to add it to spree config list of payment methods first. That can be achieved by adding the following code in your spree.rb for example:

```
Rails.application.config.spree.payment_methods << YourCustomGateway
```

better_spree_paypal_express and spree-adyen are good examples of standalone custom gateways. No dependency on spree_gateway or activemerchant required.

- Internationalization (i18n)
- View
- Asset
- Logic
- Checkout

- **Deployment**
    - Ninefold
    - Heroku
    - Shelly Cloud
    - Ansible Ubuntu Deployment
    - Manual Ubuntu Deployment
    - Deployment Options
    - Deployment Service
    - Deployment Tips
    - Requesting/Configuring SSL

- **Advanced Topics**
    - Search Engine Optimization
    - Developer Tips
    - Migrating to Spree
    - Security
    - Testing Spree Applications

- **Upgrade Guides**
    - 0.60.x to 0.70.x
    - 0.70.x to 1.0.x
    - 1.0.x to 1.1.x
    - 1.1.x to 1.2.x
    - 1.2.x to 1.3.x
    - 1.3.x to 2.0.x
    - 2.0.x to 2.1.x
    - 2.1.x to 2.2.x
    - 2.2.x to 2.3.x

Product

- Platform
- Hub
- Support
- Privacy Policy
- Terms of Service

Developers

- Overview
- Documenation
- Community
- License

Partners

- Pro Services
- Training
- Partnership Program
- Payments

About Spree Commerce

Spree Commerce is an automated enterprise solution built specifically for ecommerce. We effectively manage your operations so you can focus on serving your customers and growing your business.

Take it for a test drive

You can even create your own personal store.

Try The Demo

-
-
-
-