spreecommerce™
BEHIND THE BEST STOREFRONTS™

The Checkout Flow API - Developer Guide | Spree Commerce

Overview

The Spree checkout process has been designed for maximum flexibility. It's been redesigned several times now, each iteration has benefited from the feedback of real world deployment experience. It is relatively simple to customize the checkout process to suit your needs. Secure transmission of customer information is possible via SSL and credit card information is never stored in the database.

The customization of the flow of the checkout can be done by using Spree's `checkout_flow` DSL, described in the Checkout Flow DSL section below.

Default Checkout Steps

The Spree checkout process consists of the following steps. With the exception of the Registration step, each of these steps corresponds to a state of the `Spree::Order` object):

- Registration (Optional - only if using spree_auth_devise extension, can be toggled through the `Spree::Auth::Config[:registration_step]` configuration setting)
- Address Information
- Delivery Options (Shipping Method)
- Payment
- Confirmation

The following sections will provide a walk-though of a checkout from a user's perspective, and offer some information on how to configure the default behavior of the various steps.

Registration

Prior to beginning the checkout process, the customer will be prompted to create a new account or to login to their existing account. By default, there is also a "guest checkout" option which allows users to specify only their

email address if they do not wish to create an account.

Technically, the registration step is not an actual state in the `Spree::Order` state machine. The `spree_auth_devise` gem (an extension that comes with Spree by default) adds the `check_registration` before filter to the all actions of `Spree::CheckoutController` (except for obvious reasons the `registration` and `update_registration` actions), which redirects to a registration page unless one of the following is true:

- `Spree::Auth::Config[:registration_step]` preference is not `true`
- user is already logged in
- the current order has an email address associated with it

The method is defined like this:

```
def check_registration
  return unless Spree::Auth::Config[:registration_step]
  return if spree_current_user or current_order.email
  store_location
  redirect_to spree.checkout_registration_path
end
```

The configuration of the guest checkout option is done via Preferences. Spree will allow guest checkout by default. Use the `allow_guest_checkout` preference to change the default setting.

Address Information

This step allows the customer to add both their billing and shipping information. Customers can click the “use billing address” option to use the same address for both. Selecting this option will have the effect of hiding the shipping address fields using JavaScript. If users have disabled JavaScript, the section will not disappear but it will copy over the address information once submitted. If you would like to automatically copy the address information via JavaScript on the client side, that is an exercise left to the developer. We have found the server side approach to be simpler and easier to maintain.

The address fields include a select box for choosing state/province. The list of states will be populated via JavaScript and will contain all of the states listed in the database for the currently selected country. If there are no states configured for a particular country, or if the user has JavaScript disabled, the select box will be replaced by a text field instead.

The default “seed” data for Spree only includes the U.S. states. It’s easy enough to add states or provinces for other countries but beyond the scope of the Spree project to maintain such a list.

The state field can be disabled entirely by using the `Spree::Config[:address_requires_state]` preference. You can also allow for an “alternate phone” field by using the `Spree::Config[:alternative_billing_phone]` and `Spree::Config[:alternative_shipping]` fields.

The list of countries that appear in the country select box can also be configured. Spree will list all countries by default, but you can configure exactly which countries you would like to appear. The list can be limited to a

specific set of countries by configuring the `Spree::Config[:checkout_zone]` preference and setting its value to the name of a Zone containing the countries you wish to use. Spree assumes that the list of billing and shipping countries will be the same. You can always change this logic via an extension if this does not suit your needs.

Delivery Options

********** TODO **********

Better shipment documentation here after split_shipments merge.

****

During this step, the user may choose a delivery method. Spree assumes the list of shipping methods to be dependent on the shipping address. This is one of the reasons why it is difficult to support single page checkout for customers who have disabled JavaScript.

Payment

This step is where the customer provides payment information. This step is intentionally placed last in order to minimize security issues with credit card information. Credit card information is never stored in the database so it would be impossible to have a subsequent step and still be able to submit the information to the payment gateway. Spree submits the information to the gateway before saving the model so that the sensitive information can be discarded before saving the checkout information.

Spree stores only the last four digits of the credit card number along with the expiration information. The full credit card number and verification code are never stored in the Spree database.

Several gateways such as ActiveMerchant and Beanstream provide a secure method for storing a “payment profile” in your database. This approach typically involves the use of a “token” which can be used for subsequent purchases but only with your merchant account. If you are using a secure payment profile it would then be possible to show a final “confirmation” step after payment information is entered.

If you do not want to use a gateway with payment profiles then you will need to customize the checkout process so that your final step submits the credit card information. You can then perform an authorization before the order is saved. This is perfectly secure because the credit card information is not ever saved. It’s transmitted to the gateway and then discarded like normal.

Spree discards the credit card number after this step is processed. If you do not have a gateway with payment profiles enabled then your card information will be lost before it’s time to authorize the card.

For more information about payments, please see the Payments guide.

Confirmation

This is the final opportunity for the customer to review their order before submitting it to be processed. Users have the opportunity to return to any step in the process using either the back button or by clicking on the appropriate step in the “progress breadcrumb.”

This step is disabled by default (except for payment methods that support payment profiles), but can be enabled by overriding the `confirmation_required?` method in `Spree::Order`.

Checkout Architecture

The following is a detailed summary of the checkout architecture. A complete understanding of this architecture will allow you to be able to customize the checkout process to handle just about any scenario you can think of. Feel free to skip this section and come back to it later if you require a deeper understanding of the design in order to customize your checkout.

Checkout Routes

Three custom routes in spree_core handle all of the routing for a checkout:

```
put '/checkout/update/:state', :to => 'checkout#update', :as => :update_checkout
get '/checkout/:state', :to => 'checkout#edit', :as => :checkout_state
get '/checkout', :to => 'checkout#edit', :as => :checkout
```

The '/checkout' route maps to the `edit` action of the `Spree::CheckoutController`. A request to this route will redirect to the current state of the current order. If the current order was in the "address" state, then a request to '/checkout' would redirect to '/checkout/address'.

The '/checkout/:state' route is used for the previously mentioned route, and also maps to the `edit` action of `Spree::CheckoutController`.

The '/checkout/update/:state' route maps to the `Spree::CheckoutController#update` action and is used in the checkout form to update order data during the checkout process.

Spree::CheckoutController

The `Spree::CheckoutController` drives the state of an order during checkout. Since there is no "checkout" model, the `Spree::CheckoutController` is not a typical RESTful controller. The spree_core and spree_auth_devise gems expose a few different actions for the `Spree::CheckoutController`.

The `edit` action renders the checkout/edit.html.erb template, which then renders a partial with the current state, such as `app/views/spree/checkout/address.html.erb`. This partial shows state-specific fields for the user to fill in. If you choose to customize the checkout flow to add a new state, you will need to create a new partial for this state.

The `update` action performs the following:

- Updates the `current_order` with the paramaters passed in from the current step.
- Transitions the order state machine using the `next` event after successfully updating the order.
- Executes callbacks based on the new state after successfully transitioning.
- Redirects to the next checkout step if the `current_order.state` is anything other than `complete`, else redirect to the `order_path` for `current_order`

For security reasons, the `Spree::CheckoutController` will not update the order once the checkout process is complete. It is therefore impossible for an order to be tampered with (ex. changing the quantity) after checkout.

Filters

The `spree_core` and the default authentication gem (`spree_auth_devise`) gems define several `before_filters` for the `Spree::CheckoutController`:

- `load_order`: Assigns the `@order` instance variable and sets the `@order.state` to the `params[:state]` value. This filter also runs the "before" callbacks for the current state.
- `check_authorization`: Verifies that the `current_user` has access to `current_order`.
- `check_registration`: Checks the registration status of `current_user` and redirects to the registration step if necessary.

The Order Model and State Machine

The `Spree::Order` state machine is the foundation of the checkout process. Spree makes use of the state_machine gem in the `Spree::Order` model as well as in several other places (such as `Spree::Shipment` and `Spree::InventoryUnit`.)

The default checkout flow for the `Spree::Order` model is defined in `app/models/spree/order/checkout.rb` of spree_core.

An `Spree::Order` object has an initial state of 'cart'. From there any number of events transition the `Spree::Order` to different states. Spree does not have a separate model or database table for the shopping cart. What the user considers a "shopping cart" is actually an in-progress `Spree::Order`. An order is considered in-progress, or incomplete when its `completed_at` attribute is `nil`. Incomplete orders can be easily filtered during reporting and it's also simple enough to write a quick script to periodically purge incomplete orders from the system. The end result is a simplified data model along with the ability for store owners to search and report on incomplete/abandoned orders.

For more information on the state machine gem please see the README

Checkout Customization

It is possible to override the default checkout workflow to meet your store's needs.

Customizing an Existing Step

Spree allows you to customize the individual steps of the checkout process. There are a few distinct scenarios that we'll cover here.

- Adding logic either before or after a particular step.
- Customizing the view for a particular step.

Adding Logic Before or After a Particular Step

The state_machine gem allows you to implement callbacks before or after transitioning to a particular step. These callbacks work similarly to Active Record Callbacks in that you can specify a method or block of code to be executed prior to or after a transition. If the method executed in a before_transition returns false, then the transition will not execute.

So, for example, if you wanted to verify that the user provides a valid zip code before transitioning to the delivery step, you would first implement a `valid_zip_code?` method, and then tell the state machine to run this method before that transition, placing this code in a file called `app/models/spree/order_decorator.rb`:

```
Spree::Order.state_machine.before_transition :to => :delivery, :do => :valid_zip_code?
```

This callback would prevent transitioning to the `delivery` step if `valid_zip_code?` returns false.

Customizing the View for a Particular Step

Each of the default checkout steps has its own partial defined in the spree frontend `app/views/spree/checkout` directory. Changing the view for an existing step is as simple as overriding the relevant partial in your site extension. It's also possible the default partial in question defines a usable theme hook, in which case you could add your functionality by using Deface

The Checkout Flow DSL

Since Spree 1.2, Spree comes with a new checkout DSL that allows you succinctly define the different steps of your checkout. This new DSL allows you to customize *just* the checkout flow, while maintaining the unrelated admin states, such as "canceled" and "resumed", that an order can transition to. Ultimately, it provides a shorter syntax compared with overriding the entire state machine for the `Spree::Order` class.

The default checkout flow for Spree is defined like this, adequately demonstrating the abilities of this new system:

```
checkout_flow do
  go_to_state :address
  go_to_state :delivery
  go_to_state :payment, if: ->(order) {
    order.update_totals
    order.payment_required?
  }
  go_to_state :confirm, if: ->(order) { order.confirmation_required? }
  go_to_state :complete
  remove_transition :from => :delivery, :to => :confirm
```

we can pass a block on each checkout step definition and work some logic to figure if the step is required dynamically. e.g. the confirm step might only be necessary for payment gateways that support payment profiles.

These conditional states present a situation where an order could transition from delivery to one of payment, confirm or complete. In the default checkout, we never want to transition from delivery to confirm, and therefore have removed it using the `remove_transition` method of the Checkout DSL. The resulting transitions between states look like the image below:

********** TODO **********

State diagram

****

These two helper methods are provided on `Spree::Order` instances for your convenience:

- `checkout_steps`: returns a list of all the potential states of the checkout.
- `has_step?`: Used to check if the current order fulfills the requirements for a specific state.

If you want a list of all the currently available states for the checkout, use the `checkout_steps` method, which will return the steps in an array.

Modifying the checkout flow

To add or remove steps to the checkout flow, you can use the `insert_checkout_step` and `remove_checkout_step` helpers respectively.

The `insert_checkout_step` takes a `before` or `after` option to determine where to insert the step:

```
insert_checkout_step :new_step, :before => :address
# or
insert_checkout_step :new_step, :after => :address
```

The `remove_checkout_step` will remove just one checkout step at a time:

```
remove_checkout_step :address
remove_checkout_step :delivery
```

What will happen here is that when a user goes to checkout, they will be asked to (potentially) fill in their payment details and then (potentially) confirm the order. This is the default behavior of the payment and the confirm steps within the checkout. If they are not required to provide payment or confirmation for this order then checking out this order will result in its immediate completion.

To completely re-define the flow of the checkout, use the `checkout_flow` helper:

```
checkout_flow do
  go_to_state :payment
  go_to_state :complete
end
```

The Checkout View

After creating a checkout step, you'll need to create a partial for the checkout controller to load for your custom step. If your additonal checkout step is `new_step` you'll need to a `spree/checkout/_new_step.html.erb` partial.

The Checkout "Breadcrumb"

The Spree code automatically creates a progress "breadcrumb" based on the available checkout states. The states listed in the breadcrumb come from the `Spree::Order#checkout_steps` method. If you add a new state you'll want to add a translation for that state in the relevant translation file located in the `config/locales` directory of your extension or application:

```
en:
  order_state:
    new_step: New Step
```

The default use of the breadcrumb is entirely optional. It does not need to correspond to checkout states, nor does every state need to be represented. Feel free to customize this behavior to meet your exact requirements.

Payment Profiles

The default checkout process in Spree assumes a gateway that allows for some form of third party support for payment profiles. An example of such a service would be Authorize.net CIM Such a service allows for a secure and PCI compliant means of storing the users credit card information. This allows merchants to issue refunds to the credit card or to make changes to an existing order without having to leave Spree and use the gateway provider's website. More importantly, it allows us to have a final "confirmation" step before the order is processed since the number is stored securely on the payment step and can still be used to perform the standard authorization/capture via the secure token provided by the gateway.

Spree provides a wrapper around the standard active merchant API in order to provide a common abstraction for dealing with payment profiles. All `Gateway` classes now have a `payment_profiles_supported?` method which indicates whether or not payment profiles are supported. If you are adding Spree support to a `Gateway` you should also implement the `create_profile` method. The following is an example of the implementation of `create_profile` used in the `AuthorizeNetCim` class:

```
# Create a new CIM customer profile ready to accept a payment
def create_profile(payment)
  if payment.source.gateway_customer_profile_id.nil?
    profile_hash = create_customer_profile(payment)
    payment.source.update_attributes({
      :gateway_customer_profile_id => profile_hash[:customer_profile_id],
      :gateway_payment_profile_id => profile_hash[:customer_payment_profile_id])
    })
  end
end
```

Most gateways do not yet support payment profiles but the default checkout process of Spree assumes that you have selected a gateway that supports this feature. This allows users to enter credit card information during the checkout without having to store it in the database. Spree has never stored credit card information in the database but prior to the use of profiles, the only safe way to handle this was to post the credit card information in the final step. It should be possible to customize the checkout so that the credit card information is entered on the final step and then you can authorize the card before Spree automatically discards the sensitive data before

saving.

- **Tutorials**
  - Getting Started
  - Extensions
  - Deface Overrides
- **Source Code**
  - About
  - Navigating
  - Getting Help
  - Contributing
- **The Core**
  - Addresses
  - Adjustments
  - Calculators
  - Inventory
  - Orders
  - Payments
  - Preferences
  - Products
  - Promotions
  - Shipments
  - Taxation
- **Customization**
  - Authentication
  - Internationalization (i18n)
  - View
  - Asset
  - Logic
  - Checkout
- **Deployment**
  - Ninefold
  - Heroku
  - Shelly Cloud
  - Ansible Ubuntu Deployment
  - Manual Ubuntu Deployment

Product

Developers

Partners

- 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