# Demonstrate ability to customize shipping and payment methods

Customize checkout

## Describe how to troubleshoot payment methods.

Awesome Magento documentation - Payment Provider Gateway

Following is brief digest of official documentation.

Magento 2 replaces abstract payment method that you would normally extend with payment adapter. Where there used to be protected methods for every aspect - validate, set data, authorize, invoice, refund etc. - now it moves to separate classes. Just like with controllers where each action resides in own class. All you need to do is wire your pieces together with DI config.

Additional benefit - you can smoothly choose between simple class implementation or make a composite - split your business logic into even more fine grained pieces and mix them.

Terms:

- *payment adapter* - config-based payment method implementation. Holds operation pool, payment validator, value handler, form block type, info block type.

  Model\Method\Adapter

- *validator pool* - check country, currency, min-max amount etc. Magento 1 equivalent is `method.isApplicableToQuote()`.

  Fixed validator codes:

  - `global` - called in `payment method.validate()`. Called by quote payment.importData right after `payment method.assignData`.
  - `availability` - called in `payment method.isAvailable()`
  - `country` - called in `payment method.canUseForCountry()`
  - `currency` - called in `payment method.canUseForCurrency()`

  Gateway\Validator\ValidatorInterface

- *value handler pool* - returns configuration params like `can_void`, `can_capture`. Holds multiple value handlers by config param name and required handler `default`.

  - Gateway\Config\ValueHandlerPool
  - Gateway\Config\ValueHandlerInterface
- *command pool*

  - holds commands.

  Gateway\Command\CommandPool

- *gateway command*

  - authorize, sale, invoice, refund, cancel, void etc. Holds request builder, transfer factory, client, operation validator, result processor.

Magento 1 analogs - `method.authorize()`, `method.capture()`, `method.refund()` etc.

Gateway\CommandInterface

- *request builder*
    - given subject (quote, order or invoice) as array, prepares request parameters as array - amounts, transaction ids, cart contents etc.

  Gateway\Request\BuilderInterface

- *transfer factory* - convert request parameters into transfer object. Here you add lowest level call params - request method (get/post etc.), headers, API URL, encode body. Transfer object is built with transfer factory builder, just like you use searchCriteriaBuilder to create searchCriteria.

  Gateway\Http\TransferFactoryInterface

- *gateway client*
    - simply makes a call with given parameters, e.g. sends Curl request. Don't reinvent the wheel, you shouldn't need to implement this, just use default Zend and Soap clients, you only need to format transfer object for them.

  Gateway\Http\ClientInterface

- *response validator*
    - checks response - maybe HTTP response code, body, parse returned errors etc.

  Gateway\Validator\ResultInterface

- *response handler*
    - save response detauls in payment additional data, save transaction numbers, change order status, send email etc.

  Gateway\Response\HandlerInterface

- *form block type* - payment form only in adminhtml.

- *info block type* - payment info

    - Block\Info
        - like in M1, shows key-value strings, overwrite `_prepareSpecificInformation`.
    - Block\ConfigurableInfo - automatically shows values from payment.additional_information:
        - config.xml payment method declaration - show `paymentInfoKeys` and exclude `privateInfoKeys` - comma separated.
        - override `getLabel` to show human labels instead of raw field keys

config.xml standard fields:

- active
- title
- payment_action
- can_initialize
- can_use_internal
- can_use_checkout

- can_edit
- is_gateway
- is_offline
- min_order_total
- max_order_total
- order_place_redirect_url
- …

Gateway\ConfigInterface - reads scope config by pattern:

- getValue(field, store) - scopeConfig.getValue(pathPattern + methodCode)
- setMethodCode - used in getValue
- setPathPattern - used in getValue

## Passing user data from frontend checkout form to payment method:

Get payment information from frontend to backend

- in JS method renderer implement `getData`. default fields `method` and `po_number`. You can explicitly send `additional_data`, or all unknown keys will automatically bemoved into `additional_data` in backend.
- \Magento\Checkout\Model\GuestPaymentInformationManagement::savePaymentInformationAndPlaceOrder
- \Magento\Quote\Model\PaymentMethodManagement::set
- quote payment.importData
- event `sales_quote_payment_import_data_before`
- `method instance.isAvailable`
- method specification.isApplicable - checks one by one:
  - can use checkout - `$paymentMethod->canUseCheckout`
  - use for country - `$paymentMethod->canUseForCountry`
  - use for currency - `$paymentMethod->canUseForCurrency`
  - order total min max - between `$paymentMethod->getConfigData('min_order_total')` and `max_order_total`

  Model\Checks\SpecificationInterface
- `method instance.assignData`:
  - event `payment_method_assign_data_{$code}`
  - event `payment_method_assign_data`

## Old Magento 1 payment flow

1. get payment methods
   - isAvailable, event `payment_method_is_active`
   - method.isApplicableToQuote – check country, currency, min-max, zero total

2. save payment
    - quote.payment.importData:
        - event `sales_quote_payment_import_data_before`
        - method.assignData
        - method.validate
    - quote payment._beforeSave
        - method.prepareSave
    - method.getCheckoutRedirectUrl

### GET /V1/carts/mine/payment-information - get available methods

- \Magento\Checkout\Model\PaymentInformationManagement::getPaymentInformation
- \Magento\Quote\Model\PaymentMethodManagement::getList
- \Magento\Payment\Model\MethodList::getAvailableMethods
    - `method.isActive`
    - `method.isAvailable`
    - method specification. `isApplicable`

### POST /V1/carts/mine/payment-information - click Place order button

- \Magento\Checkout\Model\PaymentInformationManagement::savePaymentInformationAndPlaceOrder
    - payment = method, po_number, additional_information
- \Magento\Quote\Model\PaymentMethodManagement::set
    - `method.setChecks` - checkout, country, currency, min-max
    - quote payment.importData - method, po_number, additional_information
        - event `sales_quote_payment_import_data_before`
        - merge checks + additionalChecks – can set via DI
        - \Magento\Payment\Model\Checks\SpecificationFactory::create – use DI to register new checks by code
        - `method.isAvailable`
        - method specification.isApplicable - checkout, country, currency, min-max + DI registered
        - `method.assignData` – assuming Payment Adapter class:
            - event `payment_method_assign_data_{$code}`
            - event `payment_method_assign_data`
        - `method.validate` – assuming Payment Adapter class:
            - validator[ `global` ].validate

## Facade - Payment Adapter

- isActive - getConfiguredValue('active'):
    - value handler pool[ `active` ].handle(['field'=>'active'])
    - or value handler pool[ `default` ].handle(['field'=>'active'])
- isAvailable
    - isActive
    - validatorPool[ `availability` ].validate
    - event `payment_method_is_active` – can override result, same as M1
- assignData
    - event `payment_method_assign_data_{$code}`
    - event `payment_method_assign_data`
- validate – called after method.assignData in placeOrder
    - validatorPool[ `global` ].validate

- canUseForCountry
  - validatorPool[ `country` ].validate
  - called by method specification.isApplicable
- canUseForCurrency
  - validatorPool[ `currency` ].validate
- canOrder, canAuthorize, canCapture, canCaptuerPartial, canCaptureOnce, canRefundPartialPerInvoice, canVoid, canUseInternal, canUseCheckout, canEdit, canFetchTransactionInfo, canReviewPayment, isGateway, isOffline, isInitializationNeeded:
  - read from config `can_*`

*Commands:*

Commands are normally executed by `commandPool.get(name).execute()`. There's a strange opportunity to inject *command manager* - `commandExecutor` argument - that will run all commands instead. There's even a default command manager implementation - it contains command pool and runs then just the same. But this doesn't seem to be used.

- fetchTransactionInfo() - `can_fetch_transaction_information`, `fetch_transaction_information` gateway command
- initialize() - `can_initialize`, `initiailze` command
- order() - `can_order`, `order` gateway command. same as authorize + capture = sale?
- authorize() - `can_authorize`, `authorize` command
- capture() - `can_capture`, `capture` command
- refund() - `can_refund`, `refund` command
- cancel() - `can_cancel`, `can_cancel` command
- void() - `can_void`, `void` command
- acceptPayment() - `can_accept_payment`, `accept_payment` command
- denyPayment() - `can_deny_payment`, `deny_payment` command

Two more special commands: `vault_authorize`, `vault_sale`, `vault_capture`

braintree - BraintreeFacade = Model\Method\Adapter:

- BraintreeValueHandlerPool: default, can_void, can_cancel
- BraintreeCommandPool: authorize, sale, capture, settlement, vault_authorize, vault_sale, vault_capture, void, refund, cancel, deny_payment
- BraintreeValidatorPool: country

braintree_cc_vault - BraintreeCreditCardVaultFacade = Magento\Vault\Model\Method\Vault:

- `vaultProvider` = BraintreeFacade. very important, all methods proxy to this
- BraintreeVaultPaymentValueHandlerPool: default

Magento\Vault\Model\Method\Vault:

- pure class, no parent, interface \Magento\Vault\Model\VaultPaymentInterface
- proxies almost all methods: is`*`, can`*`, validate
- not implemented, throws error: initialize(), order(), refund(), cancel(), void(), acceptPayment(), denyPayment()
- assignData:
  - event `payment_method_assign_data_vault`
  - event `payment_method_assign_data_vault_{$code}`
  - original payment `method.assignData`
- authorize:

    - attach token extension attribute:
        - `customer_id` from payment.additional_information
        - `publish_hash` from payment.additional_information
        - order payment.extension attributes[ `vault_payment_token` ] = \Magento\Vault\Api\PaymentTokenManagementInterface::getByPublicHash( `publish_hash` , `customer_id` )
        - select from `vault_payment_token` where publish_hash = ? and customer_id = ?/NULL
    - commandManagerPool[ `method.code` ].executeBycode( `vault_authorize` )
- capture:
    - ONLY for `order` = `sale` payment action? Authorization transaction must not exist
    - attach token extension attribute
    - commandManagerPool[ `method.code` ].executeBycode( `vault_sale` )

### Payment Tips

Sometimes capture should do one of 3 things - sale, capture or vault_capture. Braintree makes CaptureStrategyCommand command that checks payment and calls needed capture type command.

Virtual types can extend and augment each other pretty heavily, for example BraintreePayPalSaleCommand is based on BraintreePayPalAuthorizeCommand and changes only request builder.

To make Vault work, after you receive normal payment, you save token by customer. Braintree does this in AuthorizationCommand handler chain in one of handlers. Token is simply read from response and put to order payment.extension_attributes[ `vault_payment_token` ].

Vault on frontend:

- in checkout config provider, finds all payment methods implementing Vault interface and returns codes for JS
- in checkout payment methods, adds `vault` payment method group
- get component providers \Magento\Vault\Model\Ui\TokenUiComponentProviderInterface - return params for JS component
    - Braintree registers component in DI \Magento\Vault\Model\Ui\TokensConfigProvider
- tokens = select from `vault_payment_token` where customer_id = ? and is_visible = 1 and is_active = 1 and expires_at > time
- for each token, run component providers[method.code].getComponentForToken
- window.checkoutConfig.payment.vault = [ `{$code}_{$i}` = componentConfig, braintree_cc_vault_0 = {…}, braintree_cc_vault_1 = {…}, ]

## What types of payment methods exist?

- offline
- online
- gateway
- vault - works over gateway, but executes different commands for authorize and capture. Always work in terms of previously saved token for customer.

## What are the different payment flows?

- isInitializationNeeded = true

Manually do the job in `initialize` and return custom order status and state.

- authorize, capture
- order = authorize + capture