SAP Commerce Connector 2011 v3.2

Checkout.com Connector for SAP Commerce Cloud

Checkout.com provides an end-to-end platform that helps you move faster, instead of holding you back. With flexible tools, granular data and deep insights, it’s the payments tech that unleashes your potential. So you can innovate, adapt to your markets, create outstanding customer experiences, and make smart decisions faster. The Connector for SAP Commerce Cloud (formerly Hybris) enables customers to implement a global payment strategy through a single integration in a secure, compliant and unified approach.

Release Compatibility

This release is compatible with:

- SAP Commerce: B2C Accelerator of SAP Commerce Cloud 2011. It is advised to install the latest patch version of SAP Commerce Cloud.
- Java 11.
- Checkout.com Java SDK version 3.

It is advised to use the latest release of this Connector available in GitHub.

Installation and setup

Installation of the Connector with Checkout.com payment functionality

Ensure that the version of SAP Commerce is supported for the plugin. Please view the Release Compatibility section for the current list of supported versions.

The Connector contains several extensions. Follow the following steps to include the Connector into your SAP Commerce application:

1. Unzip the supplied plugin zip file
2. Copy the extracted folders to the ${HYBRIS_BIN_DIR} of your SAP Commerce installation.
3. Run the `ant clean` command from within your bin/platform directory.
4. Copy the following lines into your localextensions.xml after. The extensions do not rely on any absolute paths so it is also possible to place the extensions in a different location (such as ${HYBRIS_BIN_DIR}/custom). Run the command `<path autoload="true" dir="${HYBRIS_BIN_DIR}/modules/checkoutcom"/>`
5. Run the commands below to install specific add-ons of the yaccelatorstorefront (replace "yacceleratorstorefront" with your custom storefront if relevant)

Add-on:

B2C: `ant addoninstall -Daddonnames="checkoutaddon" -DaddonStorefront.yacceleratorstorefront="yacceleratorstorefront"`

Optional

1. The checkoutsampledataaddon is optional, and can be installed by running the command: `ant addoninstall -Daddonnames="checkoutsampledataaddon" -DaddonStorefront.yacceleratorstorefront="yacceleratorstorefront"`
2. Run the `ant clean all` command from within your bin/platform directory.
3. Run `hybrisserver.sh` to startup the SAP Commerce server.
4. Update your running system using `ant updatesystem`

Except for setting up your hosts file, the Checkout.com Connector will work initially without any external setup needed.

The add-ons are independent and can be installed on separate server instances.

Installing the Connector using recipes

The Connector ships with a gradle recipe to be used with the SAP Commerce installer:

B2C: b2c_acc_plus_checkout_com with b2c and Checkout.com functionality. The recipe is based on the b2b_acc_plus recipe.

To use recipes on a clean installation, copy the folder hybris to your ${HYBRIS_BIN_DIR}

For local installations, the recipe generates a local.properties file with the properties defined in the recipe. You can optionally add your local.properties to the customconfig folder.

For cloud installations, generate a manifest.json that reflects different properties files per environment and aspect, languages packs and add-ons.

Install the Connector using recipes. Run the following commands:

- Create a solution from the accelerator templates and install the addons. `HYBRIS_HOME/installer$ ./install.sh -r [RECIPE_NAME] setup`
- Build and initialize the platform `HYBRIS_HOME/installer$ ./install.sh -r [RECIPE_NAME] initialize`
- Start a commerce suite instance `HYBRIS_HOME/installer$ ./install.sh -r [RECIPE_NAME] start`

Installing on SAP Commerce Cloud.

Follow the instructions below to install and deploy the Connector on SAP Commerce Cloud. The sample manifest.json included in the Connector serves as guide for the installation. Adapt your manifest.json file to include Checkout.com extensions.

The public, private and shared keys are included as properties in the manifest as placeholder. Add your keys as properties in the SAP Commerce Cloud environments.

Follow this guideline to prepare the repository for the deployment onto SAP Commerce Cloud. Include the Connector extensions in the folder `core-customize`.

Release Notes

- Connector for SAP Commerce Cloud version 2011, B2C Accelerator
- Support for SAP Commerce Fulfilment process incl. Checkout.com APIs for authorisation, capture, refund and void.

- 3DS 2.0 (PSD2). In case of a non-frictionless interaction, the user must enter additional information related to the Strong Customer Authentication (SCA). This is a hosted solution page provided by Checkout.com.
- SAP Commerce Backoffice. The connector provides specific customisations for the backoffice to ease the administration, configuration and management of all the operations related to the integration with the Checkout.com payment solution.
- SAP WCMS. The connector enables business users to add cards and APMs using CMS components.
- Cards (Visa, Mastercard, Carte Bancaire, Mada, Amex, JCB, Discover, Diners) and APM (Klarna, Fawry, Sofort, Paypal, Poli, Ideal, Alipay, Benefitpay, Bancontact, Giropay, Eps, Knet, Qpay, Multibanco, P24, oxxo, Google Pay, Apple Pay) payment methods.

Merchant guide

This section of the documentation explains the necessary steps the Merchant has to take in order to be able to integrate successfully with the Checkout.com payment solution.

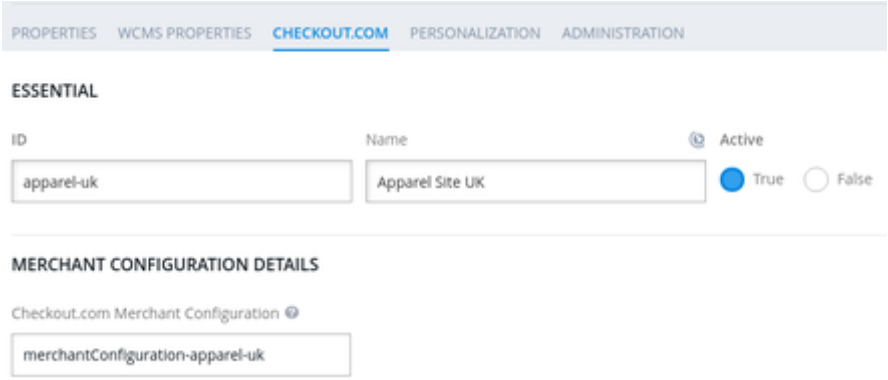
Prerequisites

Before continuing with the configuration, the Merchant is required to have:

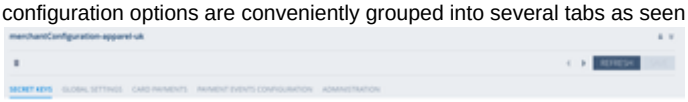
- At least a test account created on the Checkout.com website (see here)
- At least one channel and one webhook (Live) created and configured

Merchant Configuration

The SAP Commerce Connector supports multiple Checkout.com channels where a *Website* is connected to a channel.



This allows, in the case of multiple websites in SAP Commerce, to have different configurations per each channel keeping the payment transactions separated. Let's see in detail how is a Merchant Configuration defined and what are the available configuration options. The configuration options are conveniently grouped into several tabs as seen from the image below.



The configuration is an item of *CheckoutComMerchantConfiguration* type and, apart from a unique *code*, it contains the following properties which are summarised in the table below.

Tab	Property
Secret Keys Used for management of API Keys	**Public Key** Copy the key found in the Hub (see here) for the channel you are configuring. **Secret Key** same as above **Private Shared Key** Copy the key found in the channel's Webhook configured to send notifications to the platform
Global Settings General payment settings	**Environment** Must be defined as **Test** for any non-live environments (such as development, pre-production) or **Production** for production environments. **Payment Action** Use **Authorize** if you expect to perform a capture of funds separately (such as before shipping of goods) or **Auth orize and Capture** if you prefer to capture the funds immediately upon authorisation. This setting will apply to all card payments as well as to Apple Pay and Google Pay. **Review Transactions At Risk** If true, payment events coming from Checkout.com having the *Risk Flag* set to true will be created with transaction status **REVIEW**. Otherwise, the payment transaction status will be **ACCEPTED**. **Authorisation Amount Validation Threshold**

	Threshold used to determine whether the authorisation amount reflects the value of the order. This is the allowed difference of the amounts that may exist due to the way the amounts are calculated. Increase it if you observe false positives in your integration. **Billing Descriptor** In case the *Include Billing Descriptor* flag is set to true, the payment requests made will contain the Billing Descriptor Name and City having the values specified. If false, no billing descriptor will be sent.
Card Payments Settings related to card payments	**Use 3D Secure** In case it's set to **True**, the system will process payments as a 3D Secure payment. **Attempt non-3D Secure** Determines whether to attempt a 3D Secure payment as non-3D Secure should the card issuer not be enrolled.
Apple Pay Payments Settings related to Apple Pay payments	**ApplePay Configuration** Item containing the configuration details for Apple Pay. See here for **Checkout.com** ApplePay setup and configuration. **Merchant Name** • The name of the merchant shown by ApplePay **Merchant Identifier** • The merchant id registered in Apple Pay **Country Code** • The two-letter merchant's country code **Payment capabilities** • A list of supported *merchantCapabilities* as defined by the ApplePay documentation (see here) **Payment networks** • A list of *supportedNetworks* as defined by the ApplePay documentation (see here) **Private Key** • Private key of the Merchant Identity certificate created during Step 4.7 (see here) • Open the file, copy the content an paste it into the dedicated configuration property  **Certificate** • Public key (certificate) of the Merchant Identity certificate created during the Step 4.7 (see here) • Open the file, copy the content an paste it into the dedicated configuration property 
Google Pay Payments Settings related to Google Pay payments	**GooglePay Configuration** Item containing the configuration details for Google Pay **Allowed Card networks** • A list of *allowedCardNetworks* as defined by the GooglePay documentation (see here) **Allowed Card authentication methods** • A list of *allowedCardAuthMethods* as defined by the GooglePay documentation (see here) **Environment** • The environment used for processing payments. • **Test** or **Production** **Payment Gateway** • The gateway used for GooglePay (should be left to **checkoutltd**)

	Gateway merchant ID • Gateway's merchant identifier (the merchant's public key as found in the Hub) **Merchant Name** • The name of the merchant shown by GooglePay **Merchant identifier** • Merchant's Google Pay account identifier **Type** • Payment method supported by GooglePay. Use **CARD**.
Klarna Payments Settings related to Klarna payments	**Klarna Configuration** Item containing the configuration details for Klarna **Instance Id** • The ID you'll use to identify this instance of the Klarna Payments client. You should include this same instance_id for subsequent operations, like authorize, to indicate which instance the operation applies to.
Payment Events Settings related to handling of payment events	**Payment Event Types Accepted** This is a list of all the event types which will be accepted when the Webhook notifications are sent. Event types which are not in the list will be ignored. For more information about the events see here.

Payment Methods

Visibility and Availability Configuration

As summarised by the table below, the visibility (availability) of the payment methods is defined by three attributes:

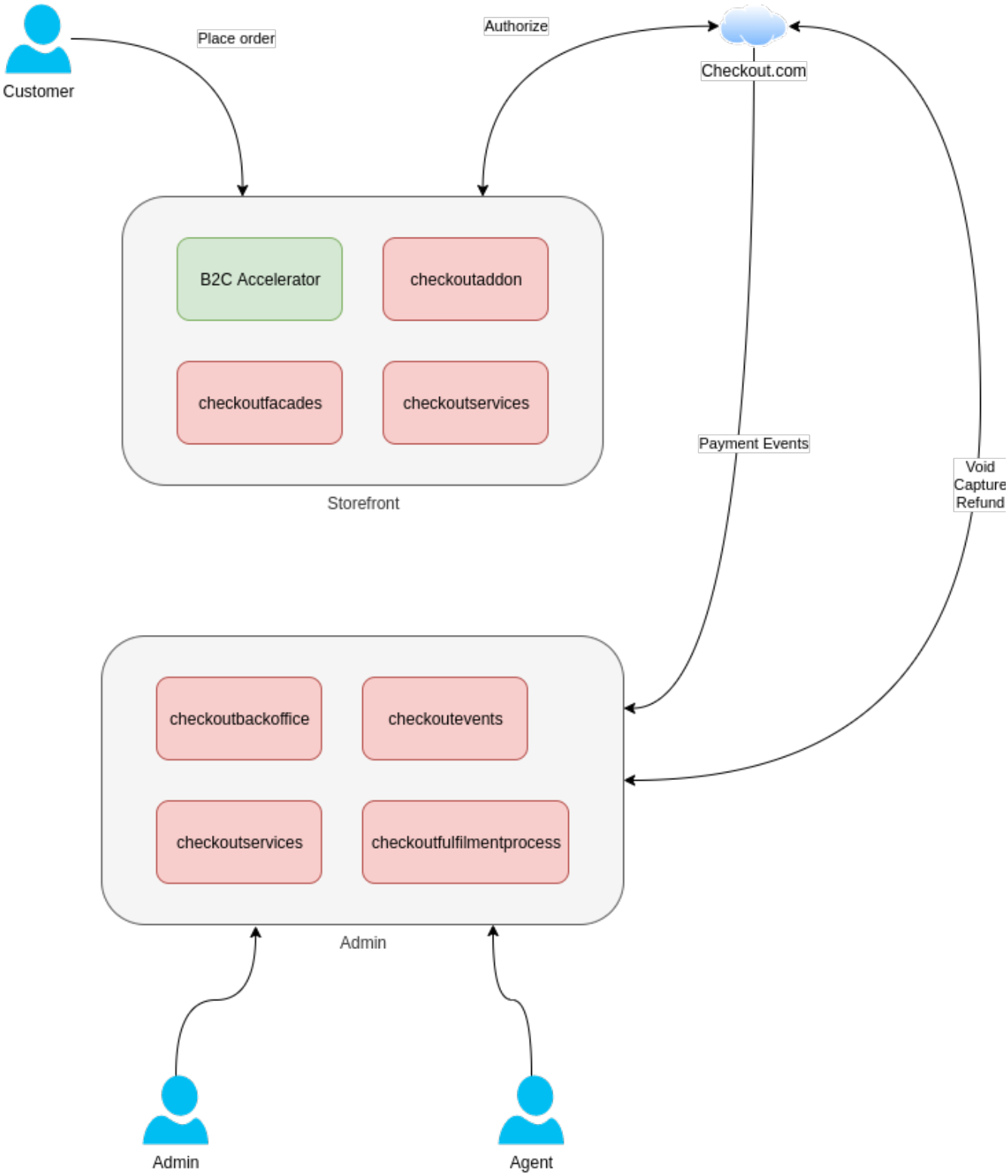
Attribute	Description
CMS Component	For a payment method to be visible, the CMS component related to the payment needs to be added to the *CheckoutCom PaymentButtonsSlot* content slot. If you have mutiple sites, this the payment should be added for each site. This is typically done by the System Integrator delivering the solution. See the *Developer Guide* for more details. Any payment methods the are globally not accepted by the Merchant (irrespective of the customer's shipping/billing country) should not be added to the CMS component.
Billing Country	Apart from card payments (available for all countries), other payment methods can be configured to have country restrictions applied. If the list is empty, there is no restriction at all. Otherwise, the payment method will be available for customers whose billing country is included in the list.
Currency	Apart from card payments (available for all countries), other payment methods can be configured to have currency restrictions applied. If the list is empty, there is no restriction at all. Otherwise, the payment method will be available for customers whose shopping cart currency is included in the list.

Payment Type	Defined in CMS Component?	Country not restricted?	Currency not restricted?	Visible
Card	YES	N/A	N/A	YES
	NO	N/A	N/A	NO
APM	YES	YES	YES	YES
	YES	NO	YES	NO
	YES	YES	NO	NO
	YES	NO	NO	NO
	NO	N/A	N/A	NO
Apple Pay	YES	N/A	N/A	Decided by Apple Pay
	NO	N/A	N/A	NO
Google Pay	YES	N/A	N/A	Decided by Google Pay
	NO	N/A	N/A	NO

Developer guide

This guide is targeted to the System Integrators who are responsible for the development of the Commerce system and payment integration. Basic SAP Commerce development principles will be covered and each of the customisation points will be explained.

Logical Architecture



There is a clear separation between the front-end and back-end extensions which are responsible for:

- Authorisation and 3D Secure request/response flows
- Receiving Webhook notifications
- Processing Webhook notifications

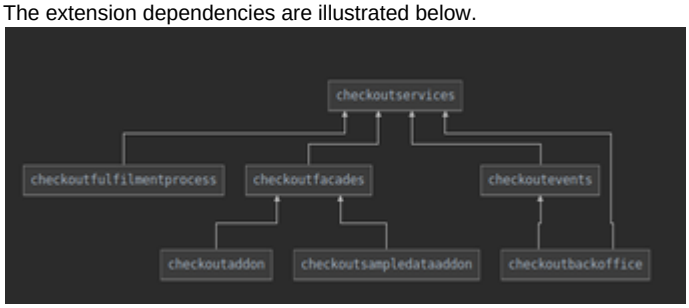
- Handling order processes including confirming of authorisation and capture
- Creating and handling return processes and refunds
- Creating and handling order cancellation processes and voids

Connector Extensions

Following the best practice SAP Commerce guidelines, the Connector comes with a number of AddOns and extensions needed to integrate with the Checkout.com payment system. The following table defines the extensions and describes their purpose.

Extension	Purpose
checkoutaddon	This is the B2C accelerator storefront AddOn which, once installed, customises the storefront checkout flow.
checkoutbackoffice	Contains Checkout.com additions to the SAP Commerce Backoffice.
checkoutevents	This extension is responsible for the management of the events received by the Webhooks defined in the Checkout.com hub. it contains the data model for the events, persistence and processing logic and the Controller which receives the Webhook events.
checkoutfacades	Extension containing custom facades used by the front-end.
checkoutfulfilmentprocess	Customised fulfilment process for the orders, returns and voids.
checkoutsampledataaddon	This AddOn, if installed, adds a full set of sample data to be used with SAP Commerce Accelerator B2C websites such as Electronics, Apparel UK and Aparel DE. It is useful to have this installed to understand how a basic payment integration works and, once understood, some of the data could be brought over and customised for the project's needs.
checkoutservices	Core extension containing the new data model, data model extensions and the core services used to integrate with the Checkout.com payment system.

The extension dependencies are illustrated below.



Extending the Connector

Whilst the Checkout.com Connector has been developed to cover most of the essential integration scenarios, some of the features could differ on a project by project basis. In order to fulfil specific project business needs, the connector can be extended and adapted to match those needs.

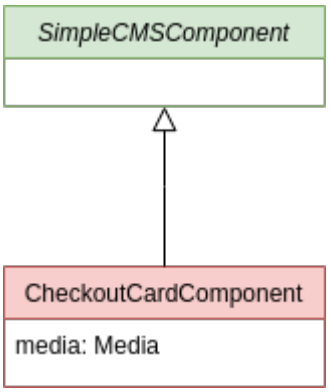
> It is strongly discouraged to change any aspect of the Connector code or data model within the connector itself. Instead, use the SAP Commerce best practices and create custom extensions depending on the Connector and make the required customisations.

The Connector provides numerous extension points on every single component which has been developed. All the beans involved have been aliased and the main methods are either public or protected facilitating their extensibility,

Data Model

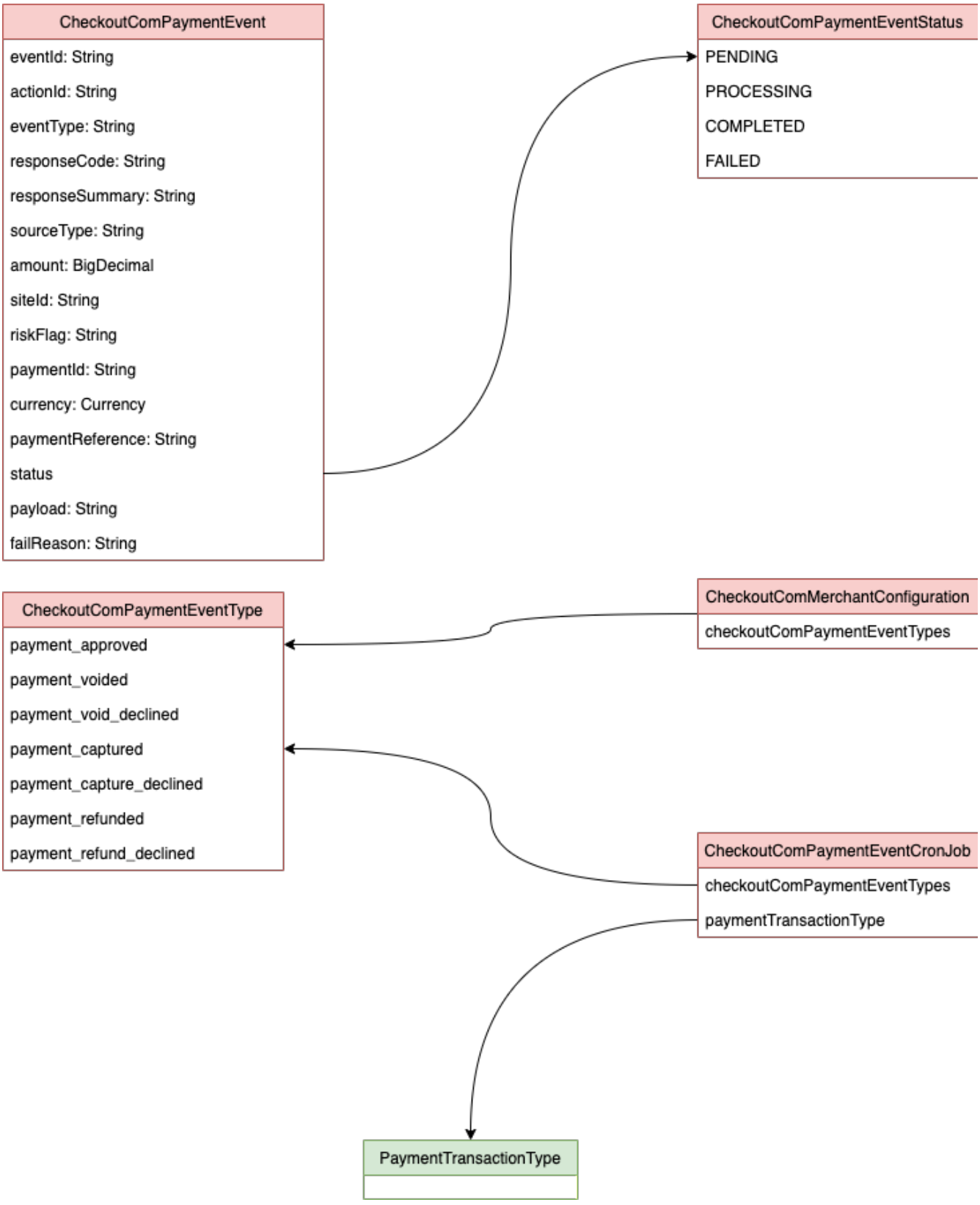
This section covers the data model changes and additions that have been made for each extension. GREEN indicates out-of-the-box items and properties (no change) and RED indicates new types and new attributes.

Extension checkoutaddon



The new *CheckoutCardComponent* type is used to define a card payment method. Once added to the *CheckoutComPaymentButtonsSlot* slot it will be visible on the Storefront

Extension checkoutevents

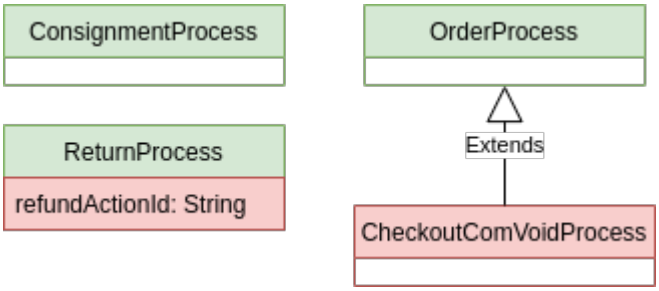


The following table summarises the items defined in this extension.

Item	Purpose
CheckoutComPayment EventType	Type of the event received (as defined here)
CheckoutComPayment EventStatus	Processing status of the event
CheckoutComPayment Event	The payment event received from the Webhook

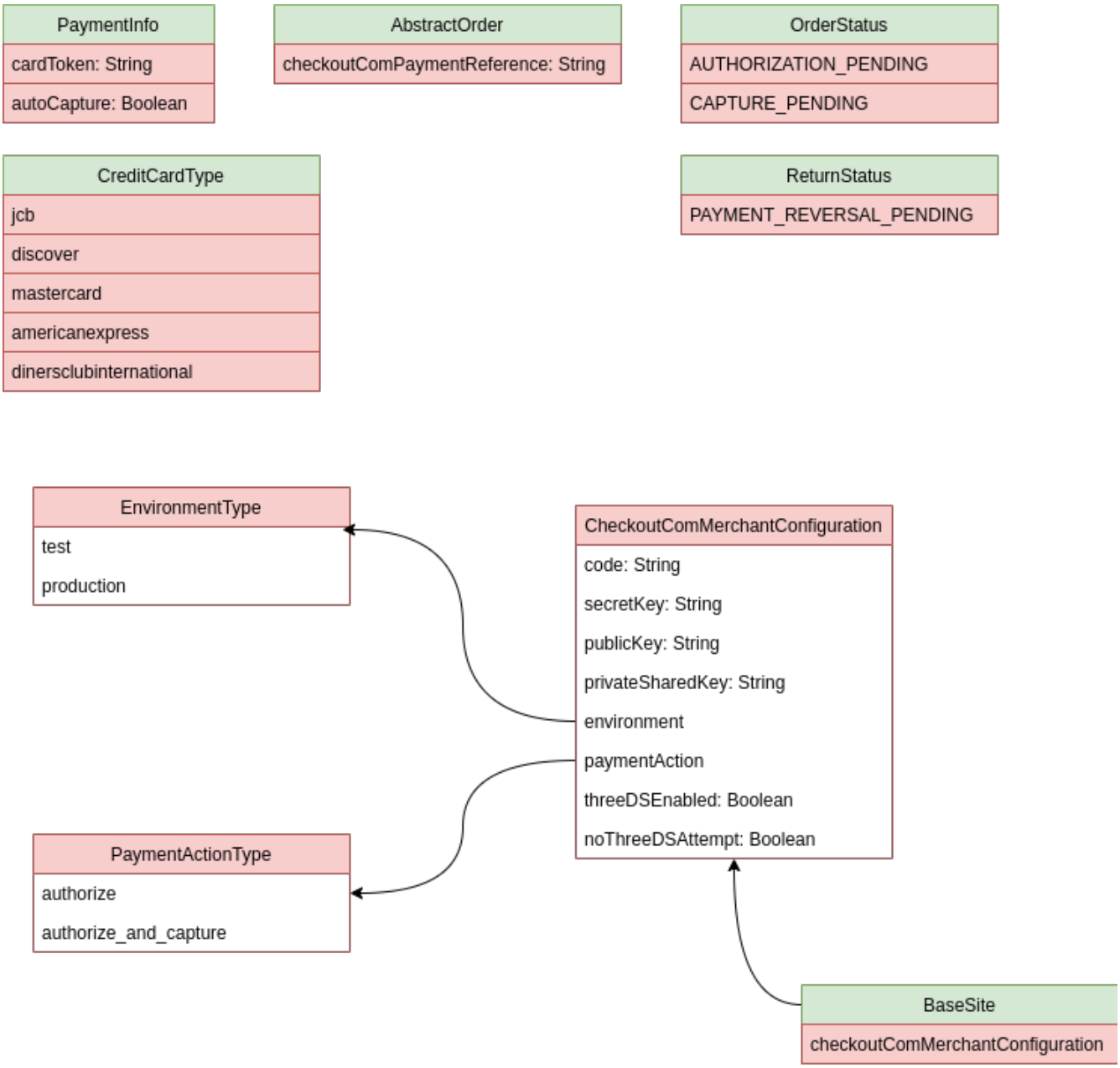
CheckoutComPayment EventCronJob	The cronjob related to the processing of the event

Extension checkoutfulfilmentprocess



Item	Purpose
ReturnProcess	Given that multiple returns (partial) are allowed, a process will be created for each return made. This will, in turn, create a Refund in Checkout.com and a unique refund action id will be identifying it.
CheckoutComVoidProcess	New type required for voiding payments.

Extension checkoutservices



Item	Purpose
PaymentInfo	Stores the generated card token and the auto-capture indicator.
AbstractOrder	The *checkoutComPaymentReference* is a generated unique reference of the cart (and the order generated from the cart) which is sent to Checkout.com. This reference is used to find the cart and/or order when required.
OrderStatus	Additional order statuses.
CreditCardType	Additional values related to the Checkout.com card types.
ReturnStatus	Additional return statuses.
CheckoutComMerchant Configuration	Stores merchant configuration options.
BaseSite	Links the SAP Commerce site to the merchant configuration.
PaymentActionType	Available payment actions

Payment Methods

Visibility and Availability Configuration

Picking up from the Merchant configuration, the main point to consider whether the merchant will support a particular payment method regardless of any country and currency restriction. If the answer is yes, then the payment method CMS component needs to be added to the dedicated content slot used in the payment form.

Please make sure this is included in your project data or imported manually into the running system. The example below is taken from the Connector's sample data addon and will make all the listed payment methods available to the site.

```
INSERT_UPDATE ContentSlot; $contentCV[unique = true]; uid[unique =
true]            ; name              ; cmsComponents(uid, $contentCV)
;; CheckoutComPaymentButtonsSlot ; "Payment Buttons" ;
checkoutComCardComponent,sofortComponent,payPalComponent,
boletoComponent,poliComponent,idealComponent,klarnaComponent,
alipayComponent,benefitpayComponent,googlePayComponent,
applePayComponent,bancontactComponent,giropayComponent,epsComponent,
knetComponent,qpayComponent,fawryComponent,multibancoComponent,
sepaComponent,p24Component,oxxoComponent
```

Payment Actions and Event Processing

Payment Actions

The Connector supports the following payment actions:

- Authorization (single)
- Capture (single, full amount)
- Refund (multiple, full or partial), only follow-on and not standalone
- Void (single, full amount)

The authorisation is done at the point of checkout based on the card token provided by Checkout.com using the Frames integration (see here). The remaining actions implement the standard SAP Commerce interfaces such as:

- *CaptureCommand*
- *FollowOnRefundCommand*
- *VoidCommand*

After the payment action has been made, a **PENDING** *PaymentTransactionEntry* of the correct type is created. The entry is then updated once the corresponding Payment Event has been received and processed. There are two exceptions to this.

The first is related to the Authorization. Once the authorization is invoked and verified as successful (during the synchronous call) the order is placed and the order process is started. The **payment_approved** event will be received and processed, creating the corresponding payment transaction and the payment transaction entry.

The second exception is related to the auto-capture payments where the capture is never initiated from the SAP Commerce system but a **pa yment_captured** event is received.

Event Processing

Checkout.com will send payment events for all the Webhooks configured in the Hub. There should be at least one Webhook created (per environment) pointing to the Webhook event receiver.

The Connector **requires** the following events to be sent:

- payment_approved
- payment_pending
- payment_declined
- payment_expired
- payment_canceled
- payment_voided
- payment_void_declined
- payment_captured
- payment_capture_declined
- payment_refunded
- payment_refund_declined

This is reflecting the merchant configuration's *checkoutComPaymentEventTypes* configuration property. If the Webhook in the Hub sends more events, they will be ignored by the event receiver.

Event Receiver

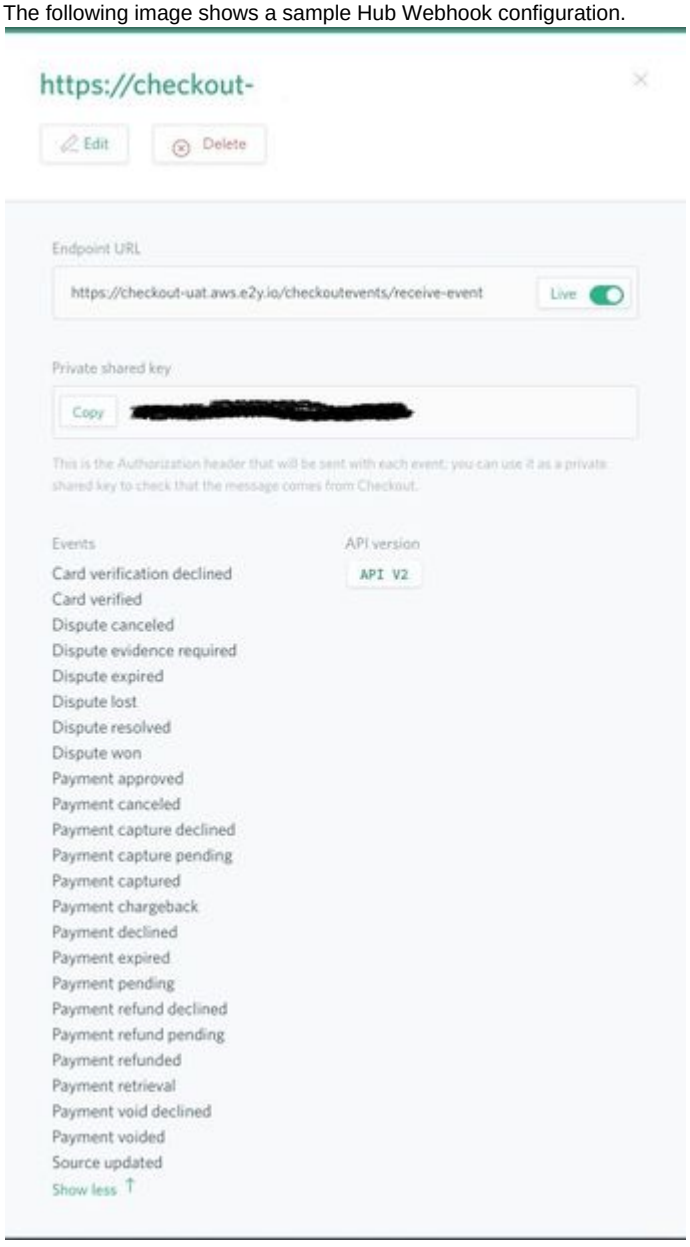
The Connector's event receiver is a Spring controller mapped as follows:

```
@Controller
@RequestMapping(value = "/receive-event")
public class CheckoutComEventController {
...
}
```

The full path to the controller is

https://<server>:<port>/chekoutevents/receive-event

The following image shows a sample Hub Webhook configuration.



The receiver will verify the *cko-signature* header sent from the webhook and, if the signature is valid, a *CheckoutComPaymentEvent* event is created. If the signature is invalid, no event is published. In any circumstance, the receiver will return an ACCEPTED HTTP Status.

Event Listener

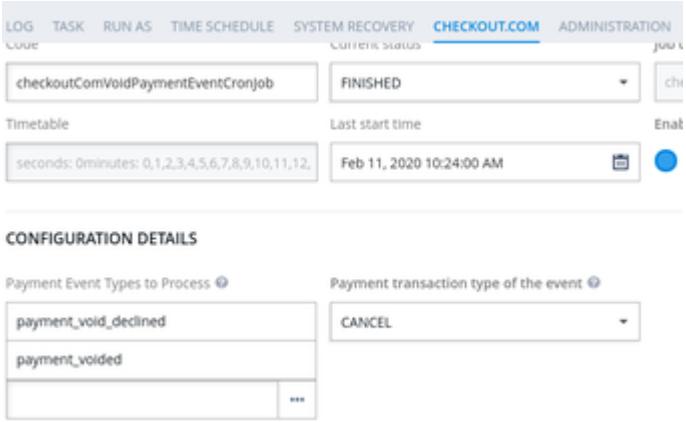
The event *CheckoutComPaymentEvent* event is a *ClusterAwareEvent* meaning that can be processed asynchronously by a listener in the cluster. This will improve the throughput and allow non-blocking processing of Webhook events. The payment event listener will check the event type and, if the merchant configuration allows it, the event will be persisted in **PENDING** state ready to be picked up by the relevant CronJob.

Event Processing Jobs

There are 4 Cron Jobs that pick pending events and process them:

1. *checkoutComAuthorisePaymentEventCronJob* - processes **authorisation** events
2. *checkoutComCapturePaymentEventCronJob* - processes **capture** events
3. *checkoutComRefundPaymentEventCronJob* - processes **refund** events
4. *checkoutComVoidPaymentEventCronJob* - processes **void** events

All the jobs have the same *checkoutComPaymentEventJob* job definition and the difference is in the configured payment event types to be processed and the corresponding payment transaction type created by the event (as shown in the image below).



Sample data addOn provides the trigger configuration for all the jobs to run every minute but it should be evaluated on case-by-case basis. For instance, a commerce site could receive authorisations more frequently than captures, and captures more frequently than voids and refunds.

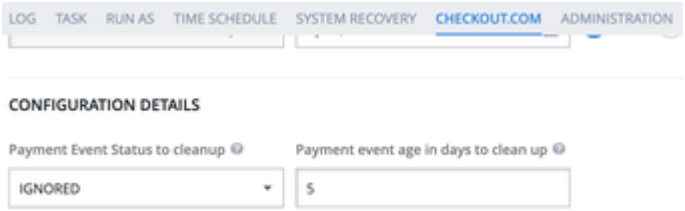
Event Cleanup Jobs

Payment Events stored in the system can end in the following statuses:

- COMPLETED
- PENDING
- FAILED
- IGNORED

Normally, pending events are meant to be processed but, occasionally some events (such as payment_pending) are triggered before the order has been placed. This is true, particularly for APM payments. If the customer does not place the order, the system will not find an order for that event and will not be able to process it.

For each event processing status, there is a corresponding cleanup job defined in the system. All the jobs have the same *checkoutComPaymentEventCleanupJob* job definition and the difference is in the configured payment event status to be processed and the maximum age (in days) for the event to be picked up and deleted (as shown in the image below).



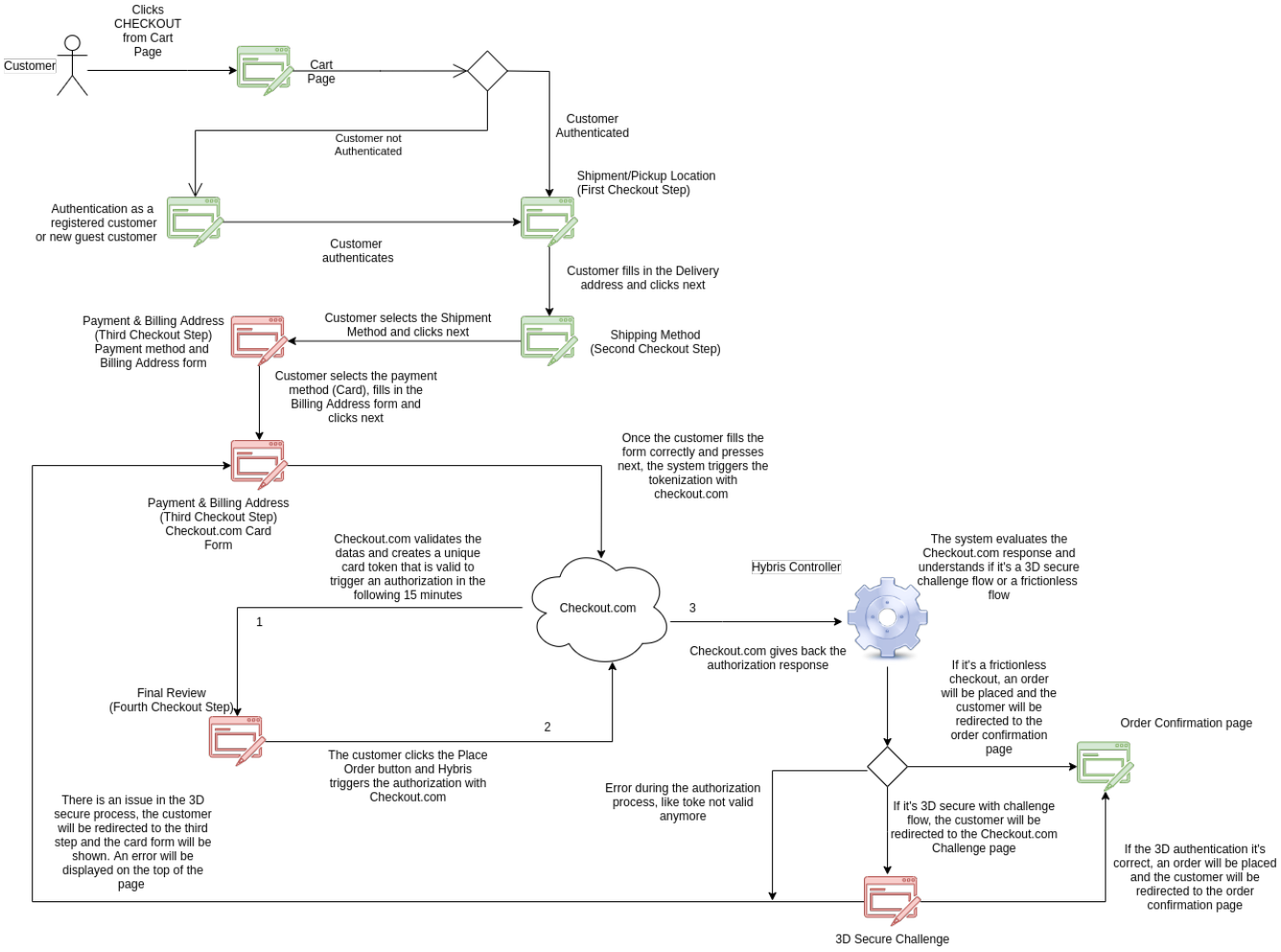
There are 4 Cron Jobs that pick events based on the status and removes them:

1. *checkoutComCompletedPaymentEventCleanupCronJob* - deletes **COMPLETED** events older than 5 days
2. *checkoutComPendingPaymentEventCleanupCronJob* - deletes **PENDING** events older than 30 days
3. *checkoutComFailedPaymentEventCleanupCronJob* - deletes **FAILED** events older than 30 days
4. *checkoutComIgnoredPaymentEventCleanupCronJob* - deletes **IGNORED** events older than 5 days

The *ageInDaysBeforeDeletion* values are only suggested values. Please adapt them to the nature of the business.

Sample data addOn provides the trigger configuration for all the jobs to run on a daily basis.

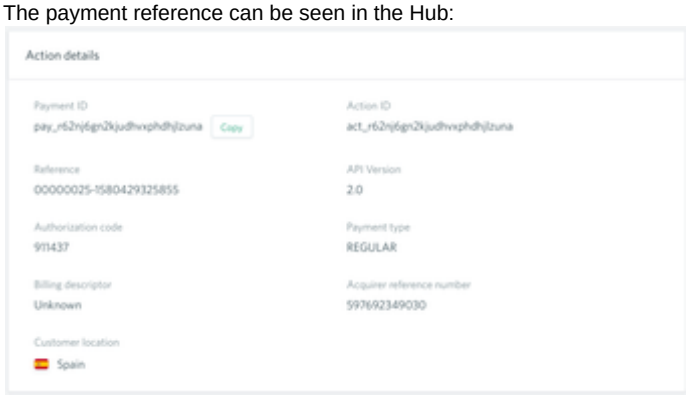
Checkout Process



Payment Reference

The payment reference is a unique, generated value that is assigned to a cart. Once the checkout process is completed, the same value will be transferred to the order. This value is a vital piece of information that is used by the Connector to marry the payment in Checkout.com to the cart/order in SAP Commerce.

The payment reference can be seen in the Hub:



And in the events received from the webhooks (example of a payment capture event):

```
{
    "id": "evt_twqujok4i7wuhlc2t7ol2htz7u",
    "type": "payment_captured",
    "created_on": "2020-01-28T14:05:07Z",
    "data": {
        "action_id": "act_hdvlb3wuquwebi5drbg73riyyy",
        "response_code": "10000",
```

```
        "response_summary": "Approved",
        "amount": 10784,
        "metadata": {
            "site_id": "electronics",
            "Udf5": "hybris 1905.4 extension develop"
        },
        "processing": {
            "acquirer_transaction_id": "9782436706",
            "acquirer_reference_number": "991602325351"
        },
        "id": "pay_3fm7urs6lhsenkvv2yybiruqdq",
        "currency": "USD",
        "processed_on": "2020-01-28T14:05:07Z",
        "reference": "00002000-1580219992786"
    },
    "_links": {
        "self": {
            "href": "https://api.sandbox.checkout.com/events
/evt_twqujok4i7wuhlc2t7ol2htz7u"
        },
        "payment": {
            "href": "https://api.sandbox.checkout.com/payments
/pay_3fm7urs6lhsenkvv2yybiruqdq"
        }
    }
}
```


The payment reference generation is defined by the *CheckoutComPaymentReferenceGenerationStrategy* interface and the connector provides a default implementation in the *DefaultCheckoutComPaymentReferenceGenerationStrategy* class.

The default implementation uses the *AbstractOrder's* code and it adds the timestamp as a suffix as shown in the above-mentioned examples. This approach works well in environments that are frequently initialised (such as local, CI and others) as it avoids generating clashing values. For environments that are not initialised (like pre-prod or production), it is recommended to have a more robust implementation such as **GUID** or similar.

To do that, re-alias the *checkoutComPaymentReferenceGenerationStrategy* alias in spring and provide your own implementation (using spring profiles).

Pay by Card

The tokenisation process is triggered by the card component form submit. When the customer clicks ***Next*** button, the form is submitted to Checkout.com. The javascript uses the public key of the current merchant which is retrieved from the merchant configuration.

Once the token is received from Checkout.com, the form containing the token is automatically submitted and validated. If the form is valid, the system generates the payment info which contains the payment token. A *PaymentInfoModel* will be created containing *cardToken* that will be needed for the authorization call. If then there is an error in the process, as described in the flow, the payment info will be deleted from the corresponding cart.

When the customer clicks the ***Place Order*** button in the final review step, the authorization will be triggered with Checkout.com. The system receives back the response and evaluates if the payment has been approved (frictionless 3DS flow) and proceeds with the place order process or, if pending (3DS challenge flow), redirects to the external 3D Secure challenge page. The 3D Secure success and failure URLs are set during the payment authorisation call.

In case of success, Checkout.com sends a **cko-session-id** as request parameter, which is used by the Connector to get the payment details from Checkout.com and the details to the session cart. In case of a match, the order will be placed and the customer will land on the Order Confirmation page. Otherwise, the customer will be redirected to the homepage.

Clustered environments without session replication

It's quite common to have a customer-facing cluster without a full session replication. The 3DS redirect is a browser redirect meaning that the browser will be communicating with the same front-end node (assuming the sticky session is configured properly). In some cases (such as node failure) the customer could be redirected to a different node and there would be no cart in the session

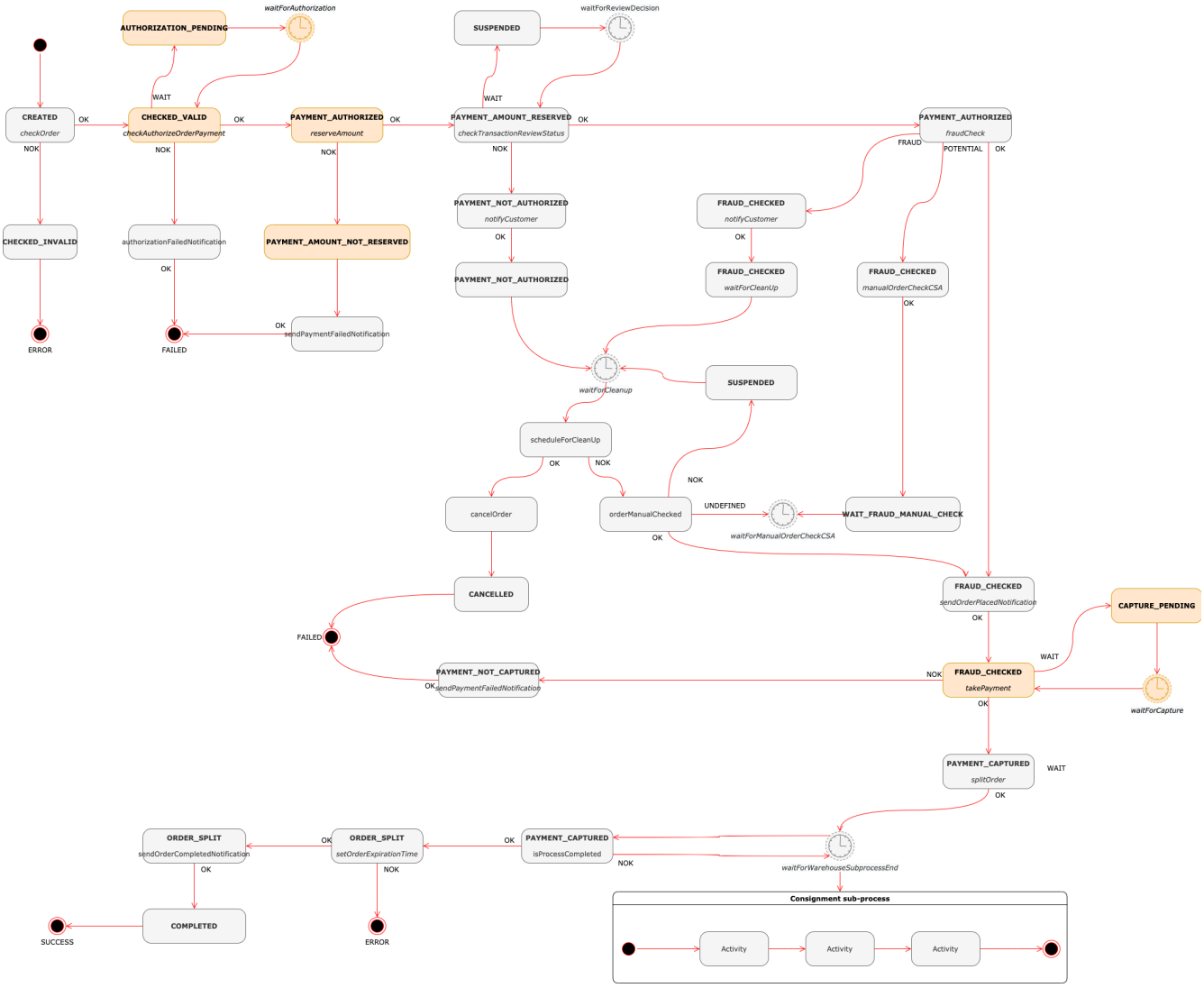
to match. The connector does not cover this scenario as it is depending on the way the commerce site is built. In case it's needed, the cart can be recovered by performing a query (cart is persisted) based on the payment reference but the place order function has to be customised too as it relies on the cart being in the session.

Alternative Payment Methods (APM)

APMs can be divided into two groups when referring to payment flow; Direct and Redirect.

For direct APMs, such as Klarna, Fawry and Benefitpay, payments are authorised directly on the platform using Checkout.com payment APIs. Redirect payment methods (such as PayPal and others) follow a flow similar to 3D Secure for card payments in which the customer is redirected to the third party site to complete the payment.

Fulfilment Process



The diagram above represents a state diagram of an order during the *order-process* which is started when the order is placed. The grey boxes are out-of-the-box SAP Commerce states and events whereas the coloured ones are related to the Connector customisations.

NOTE: The order process is the standard B2C order process which is delivered using the *b2c_acc_plus* installer recipe (see here). When defining your custom order process, plug in the custom actions into your own process definition.

There are two main areas of customisation; Authorisation and Capture.

Authorisation

When the order reaches the state of **CHECKED_VALID** the transition will be handled by the custom *checkAuthorizeOrderPayment* action. If the order has no payment transaction entry of type **AUTHORIZATION**, the process will set the order state to **AUTHORIZATION_PENDING** and wait for the related payment event received from the Webhook to be processed. In case the event has been processed before the order process reaches this stage, the process will continue as normal. Finally, if the authorisation has been processed but the payment has been declined, the process will end in a failed state and a **REJECTED** transaction entry will be added to the transaction.

Reserving Order Amount

Once the process is past the authorisation, a customised *reserveOrderAmount* action will check whether the order amount matches the authorised amount received from **Checkout.com**. If the amounts do not match, it means that some sort of unexpected order manipulation has happened between the initial order authorisation and the authorisation confirmation (intrinsic for payment flows leaving the website before coming back for confirmation). In this case, the action terminates putting the order process in **ERROR** state. Furthermore, a *PaymentFailedEvent* is published and the order status is set to **PAYMENT_AMOUNT_NOT_RESERVED**.

NOTE: The service responsible for checking the amount is *CheckoutComPaymentTransactionService* which uses the *isAuthorisedAmountCorrect* method.

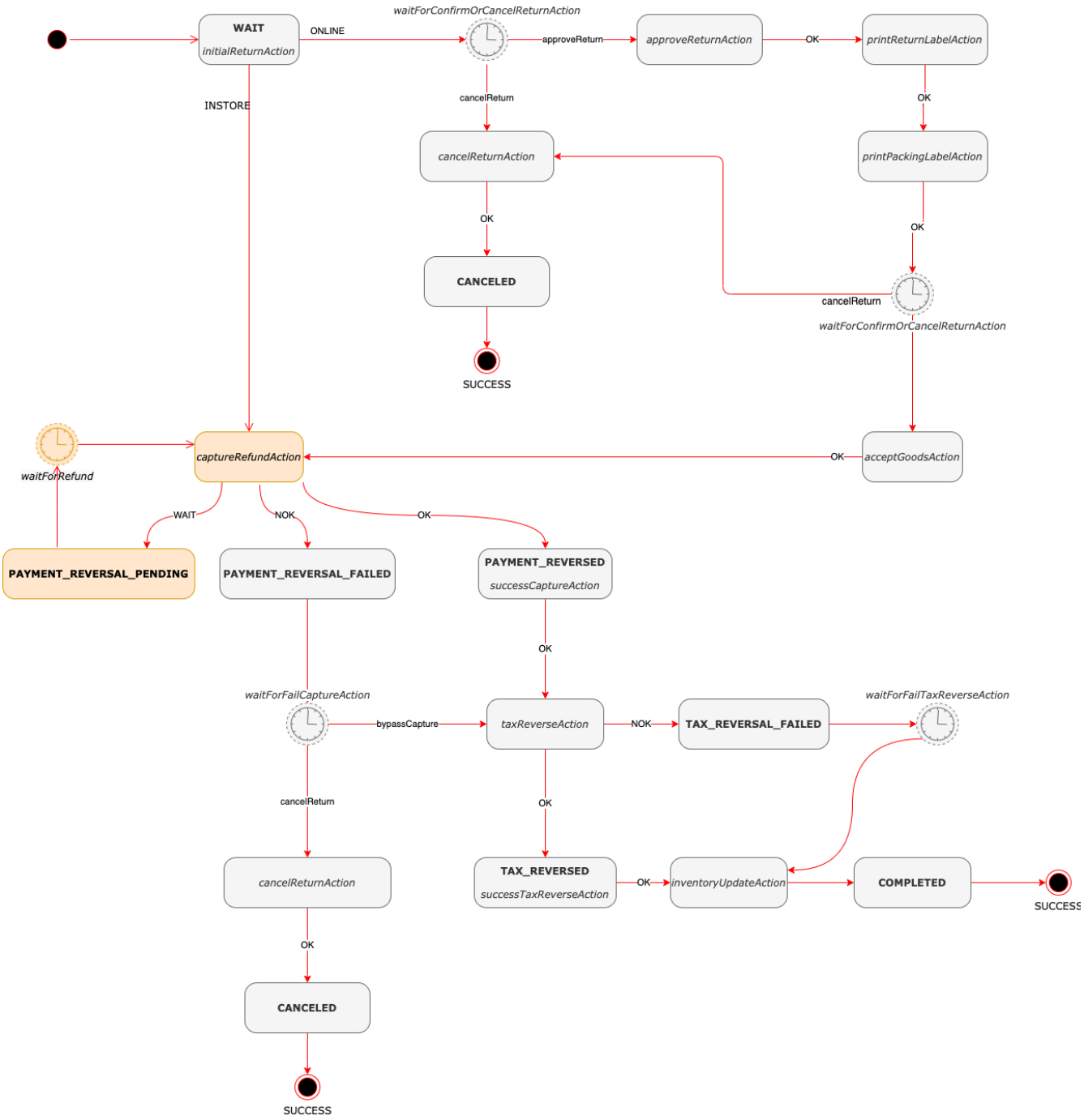
Risk

If the payment notification arriving from has been marked as RISK due to various risk factors (see here), the Connector will decide the course of action based on the merchant configuration setting for *reviewTransactionsAtRisk*. In case the value is enabled then the Connector will mark the payment transaction entry as **REVIEW**. The order process will continue normally as per default behaviour and reach the **SUSPENDED** state because of the *checkTransactionReviewStatus* action. The review process is defined on project basis and is not covered by the Connector. Otherwise, the payment transaction entry will be created as **ACCEPTED** and the order process will continue.

Capture

When the order reaches the state of **FRAUD_CHECKED**, the system will be ready to take the payment (capture). This will be handled by the custom *takePayment* action. If the order has no payment transaction entry of type **CAPTURE**, the action will try to capture the payment unless the payment type is auto-capture. The process will set the order state to **CAPTURE_PENDING** and wait for the related payment event received from the Webhook to be processed. In case the event has been processed before the order process reaches this stage, the process will continue as normal. Finally, if the capture has been processed but the payment has been declined, the process will end in a failed state and a **REJECTED** transaction entry will be added to the transaction.

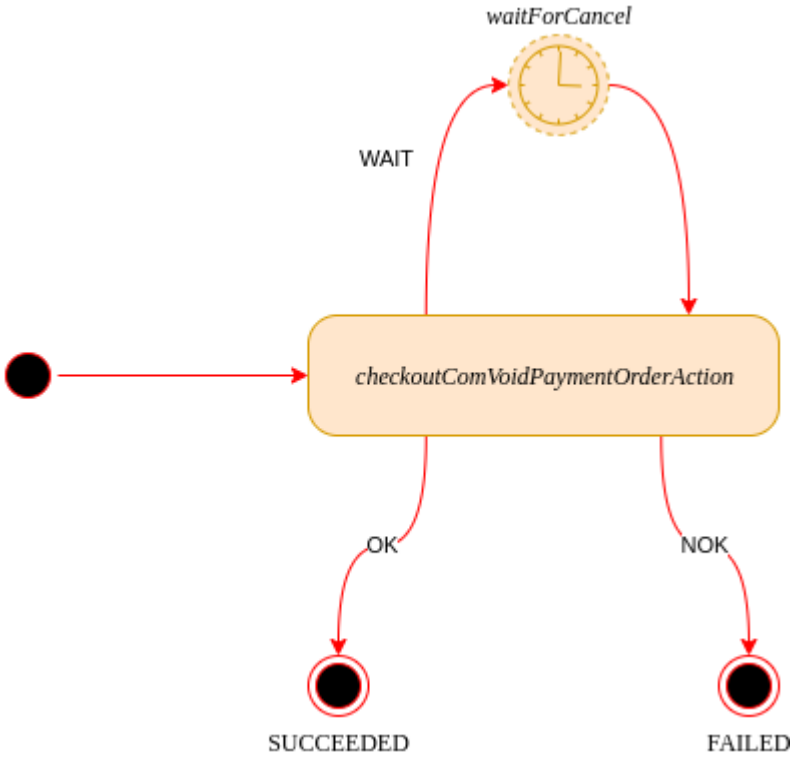
Return Process



This is the typical return process defined by the B2C accelerator in the *return-process.xml* definition file. The Connector provides an event listener *CreateReturnEventListener* which will start the return business process. The capture refund action has been customised in order to create a **PENDING** follow-on refund transaction and wait for the refund confirmation event before continuing with the remaining out-of-the box accelerator refund process.

The refund action *DefaultCheckoutComCaptureRefundAction* only provides a basic refund calculation for demonstration purposes. The class should be extended and realiased with a custom component that will override *gerRefundAmount* method and consider the business requirements (shipping costs, promotions, etc.)

Cancellation Process



The cancel process can be triggered from the storefront or from the backoffice custom support.

From the storefront, a logged-in customer can cancel an order from the order history page. The cancel button will be shown only if the order has been already authorised and if the capture has not been done yet. A strategy name CheckoutComPaymentStatusOrderCancelDenialStrategy has been created in order to implement this rule for storefront and backoffice (for backoffice the button is always present but will be disabled).

The plugin does not allow partial cancellations. This rule has been implemented with the strategy CheckoutComPartialOrderCancelDenialStrategy. So if the customer/agent triggers a partial void, will receive an error message at the end of the journey.

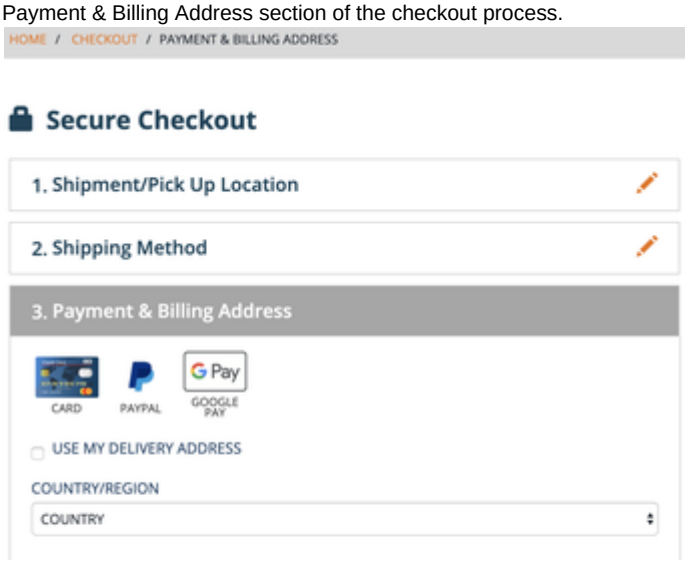
Once the cancellation is triggered, the order goes automatically in **CANCEL** status and an event called **CancelFinishedEvent** is triggered.

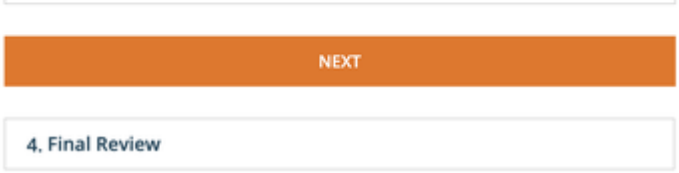
A listener will intercept this event and will create/triggers the **void-process**. This is a custom business process that can be found in the type model as **CheckoutComVoidProcess**.

Once the process starts, the only action called **CheckoutComVoidOrderAction** checks if the order has been already voided, and if yes will set the process to **SUCCEEDED**. If not the action triggers the cancel with Checkout.com (with the custom command CheckoutComVoidCommand) and evaluates the created **CANCEL** transaction entry. If the transaction entry is **ACCEPTED**, it will set the process to **SUCCEEDED**. If the transaction entry is **ERROR**, it will set the process to **FAILED**. If the transaction entry is **PENDING**, it will set the process to **waitFor_CANCEL** and it will wait until the event created from the void event processing.

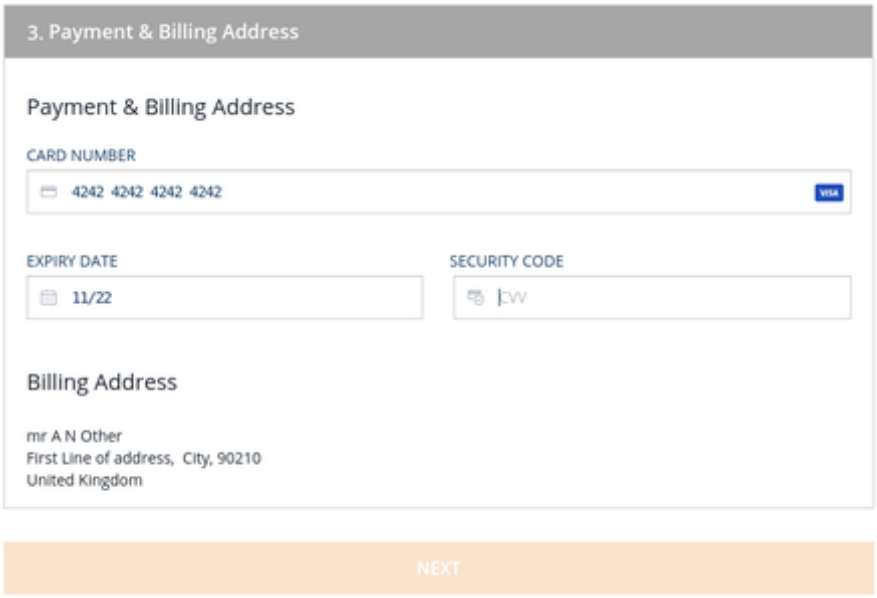
Storefront Customisation

The AddOn is compatible with the Responsive version of the B2C Accelerator storefront. The changes the AddOn brings are related to the Payment & Billing Address section of the checkout process.



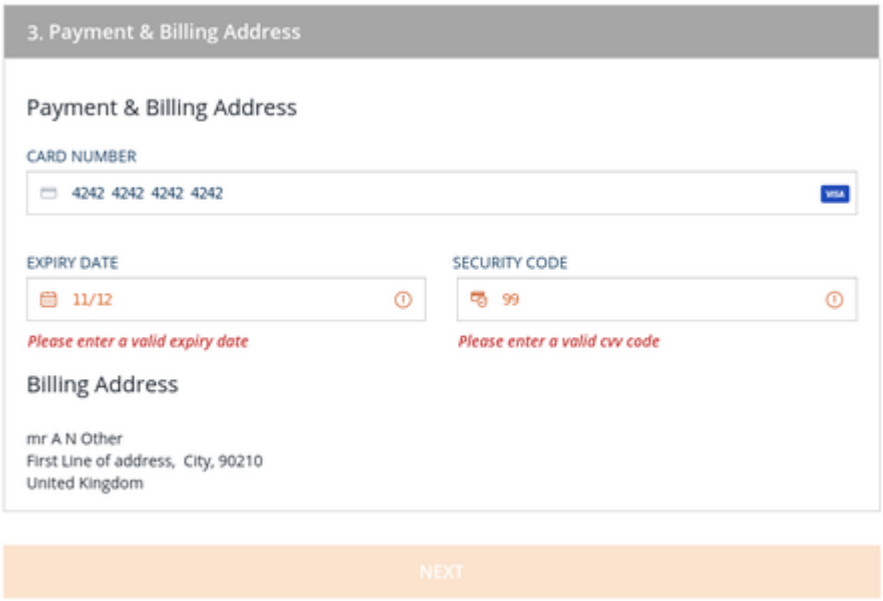


The section will display the allowed payment methods based on the Site configuration. Once the payment method is selected and the billing address is set, the user is shown the payment forms where the card details can be entered.



The user interaction is seamless as the data is entered in the form which is integrated with Checkout.com using the Frames (see here) integration for a fully compliant PCI solution.

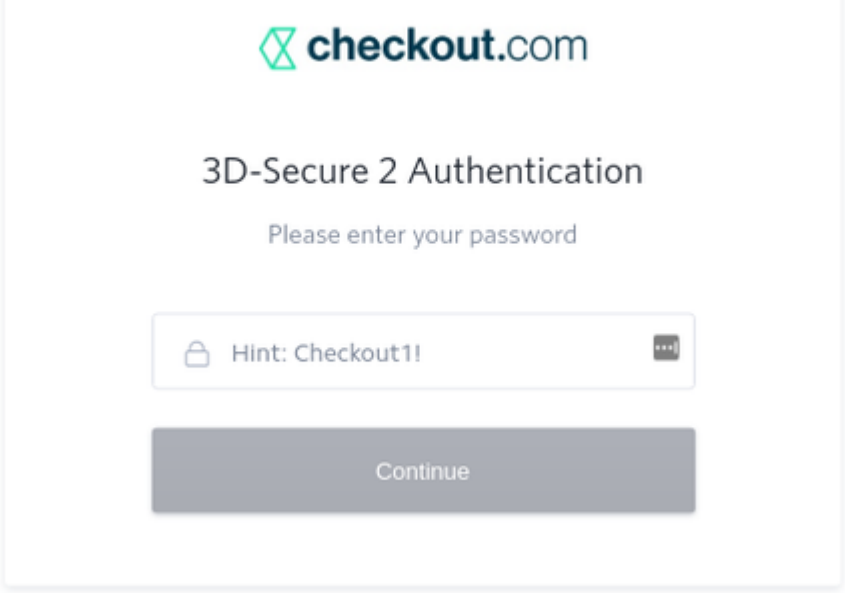
Once the form is complete and valid, the **Next** button is enabled and the user can carry onto the final step. In case of errors, the Frames integration will display the errors within the form.



The **Final Review** step is the last step of the checkout process. The card has been tokenised at this point and, once the user accepts the Ts & Cs, the tokenised card will be used to perform the payment authorisation.

3D Secure

The AddOn is compliant with the 3DS 2.0 (PSD2) directive. In case of a non-frictionless interaction, the user must enter additional information related to the Strong Customer Authentication (SCA). This is a **hosted solution** page provided by Checkout.com.

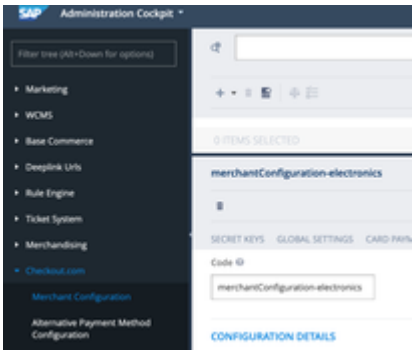


Upon a successful customer authentication, the user is redirected to the confirmation page.

GooglePay 3D Secure

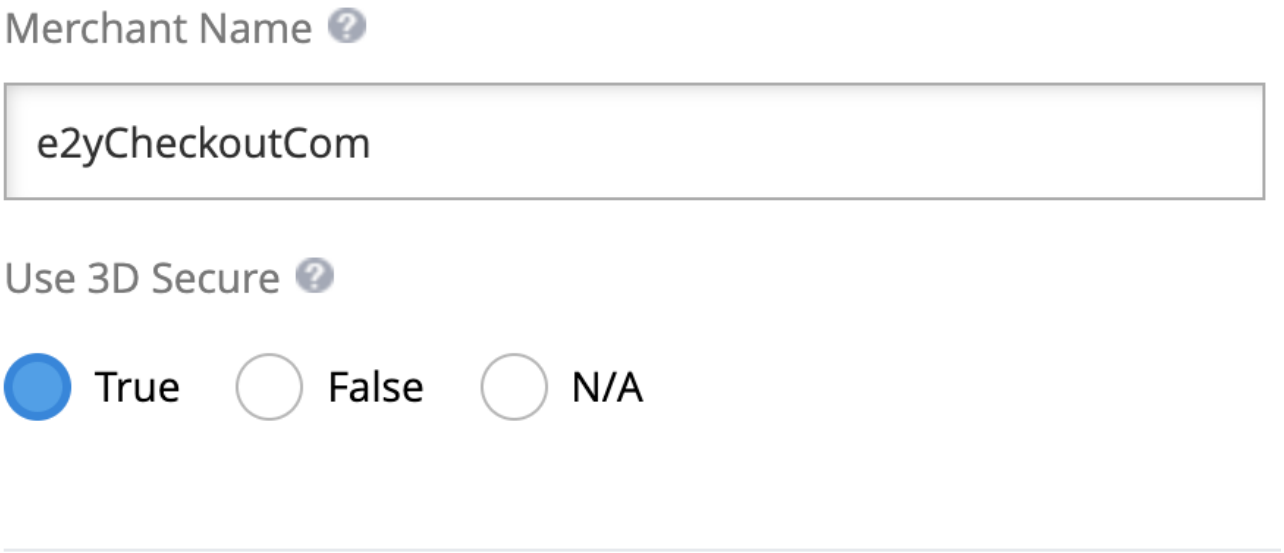
To enable 3D Secure for Google Pay, please contact your Customer Success Manager on Checkout. Once the 3D Secure has been enabled on checkout merchant we will need to enable it in googlePayConfiguration on hybris side as well.

Go to Merchant Configuration in Checkout.com area in backoffice and select the Merchant configuration for which we want to enable 3D Secure:



Go to the GooglePay Payments tab and double click the GooglePayConfiguration Object:

Set the Use 3D Secure to true.



When placing the order will googlePay will be redirected to the 3D Secure page , same flow as 3D Secure for Card payments

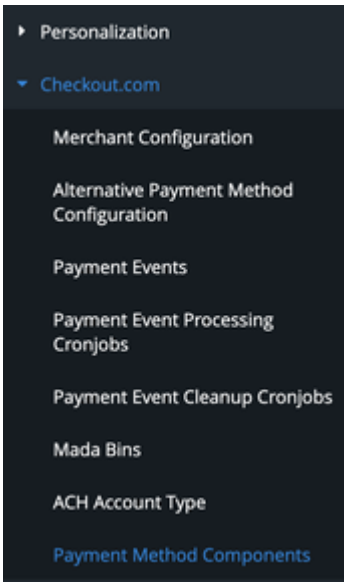
Note: that the Checkout 3D Secure flag and Hybris 3D Secure flag need to be the same value, otherwise payment will fail.

Backoffice Customisation

The connector provides specific customisations for the backoffice to ease the administration, configuration and management of all the operations related to the integration with Checkout.com payment solution.

Custom Node

A **Checkout.com** node is added which groups the most commonly required functionalities.

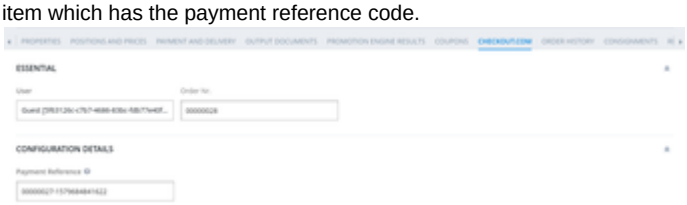


Node	Purpose
Merchant Configuration	Contains the merchant configuration items which contain the integration keys and general settings necessary to communicate and integrate with the Checkout.com payment solution.
Alternative Payment Method Configuration	Contains the list of APM configuration items where the merchant can define applicability rules such as countries and currencies the APM will be available for.
Payment Events	Contains the list of payment events received via Webhooks (see here) from Checkout.com and their processing status.

Payment Event Processing Cronjobs	Contains the cronjobs which process the payment events received.
Payment Event Processing Cronjobs	Contains the cronjobs which clean-up the processed payment events.
Mada Bins	Contains the list of card bins which identify a payment as MADA.
Payment Method Components	CMS components for the payment methods

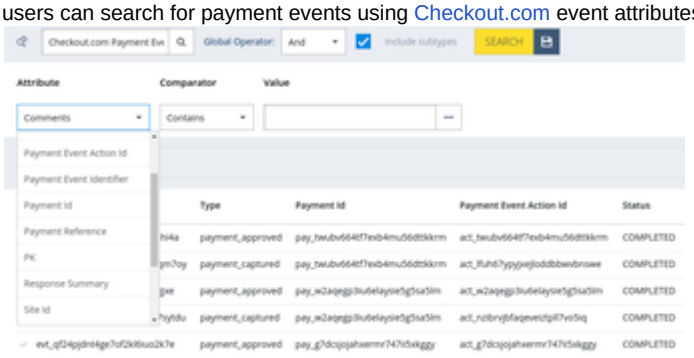
Tabs

Where needed, a CHECKOUT.COM tab will group the custom item type properties for easier access. The following image depicts the Order item which has the payment reference code.



Advanced Search

Relevant custom search attributes have been added to enable backoffice users to perform targeted searches. As shown in the image below, users can search for payment events using Checkout.com event attributes.



Connector Limitations

- Customer service: There are no new features added to the Customer Services as such but only logic customisations related to what actions can or cannot be done by the agent (void, cancel…).
- ASM (MOTO) payment is not supported
- A single payment method can be used to complete the purchase
- Only full order cancellations are allowed
- No multiple captures are possible (only full amount)
- There is no Collect-in-Store support out-of-the-box
- Klarna discount strategy to be implemented
- Klarna only supports GROSS prices
- No synchronisation of payment actions performed in the Hub

Spartacus

Checkout.com now has a connector for Spartacus 4.2. This includes new features:

- Show first name + last name as the card account holder
- Fix for ApplePay transaction status

Requirements

Requirements are dictated by Spartacus. The connector is built and tested on version 4.2.0 of Spartacus. Make sure your system is setup to the right requirements:

- Angular CLI: Version 10.1 or later, < 11.
- Node.js: The most recent 12.x version is recommended, < 13.
- Yarn: Version 1.15 or later.
- Spartacus 4.2.x
- Requires Spartacus Feature Modules: `checkout` and `order`

Installation

Unzip the archive in your storefront and add the connector and the translations to your `package.json`

```
"dependencies": {
  ...
  checkout-spartacus-connector": "file:path/to/connector",
  checkout-spartacus-translations": "file:path/to/translations",
  ...
}
```

To install the connector, you have to append the feature module and translations to your SpartacusConfigurationModule in your Angular storefront App.

```
import { checkoutComTranslationChunkConfig, checkoutComTranslations }
from 'checkout-spartacus-translations';

@NgModule({
  providers: [
    ....,
    provideConfig({
      featureModules: {
        CheckoutComComponentsModule: {
          module: () => import('checkout-spartacus-connector').then(m
=> m.CheckoutComComponentsModule),
          cmsComponents: [
            'CheckoutPaymentDetails',
            'CheckoutPlaceOrder',
            'OrderConfirmationThankMessageComponent',
            'OrderConfirmationOverviewComponent',
            'OrderConfirmationItemsComponent',
            'OrderConfirmationShippingComponent',
            'OrderConfirmationTotalsComponent',
            'OrderConfirmationContinueButtonComponent',
            'CheckoutReviewOrder',
            'AccountOrderDetailsItemsComponent',
            'AccountOrderDetailsShippingComponent',
```

```
          ],
        }
      }
    } as CmsConfig), provideConfig({
    i18n: {
      resources: checkoutComTranslations,
      chunks: checkoutComTranslationChunkConfig,
      fallbackLang: 'en'
    },
  } as I18nConfig), provideConfig({
      checkout: {
        guest: true // not required, but we support guest checkout
      }
    } as CheckoutConfig),
    ...
  ]
```

Being a feature module, the code will only be loaded the moment we enter the third step of the checkout (Payment Details). The translations can't be be lazy loaded, so this is why it has been moved to separate node module.

At the bottom of the body of your `index.html`, you will have to add the Frames script. Frames will log customer behaviour while browsing the website.

```
<body>
  <app-root></app-root>
  <script src="https://cdn.checkout.com/js/framesv2.min.js"></script>
</body>
```

Guest checkout

We support Guest checkout and can be configured using the configuration

Checkout OrderConfirmationGuard

Checkout.com makes use of a redirect to external pages to complete the payment process. After the payment is authorised, or not authorised, the client is sent back to the order confirmation page, given 2 request parameters. We have replaced the OOTB `OrderConfirmationGuard` to make sure the order is placed after returning back to the storefront. If you have modified the guard yourself, you will have to extend our `CheckoutComCheckoutGuard` and apply this guard to the following components:

- OrderConfirmationThankMessageComponent
- OrderConfirmationItemsComponent
- OrderConfirmationTotalsComponent
- OrderConfirmationOverviewComponent

Translations

We have created a default set of translations for labels and error messages shown in the connector. If you want to override these, you can do so in the translations config of your App.

Overriding setDeliveryAddress endpoint

We have overridden the setDeliveryAddress endpoint (using `defaultOccCheckoutComConfig`) because we also have to set the payment address in step 1 of the checkout. This is required for displaying the available payment methods in Step 3 - Payment details.