STRIPE INTEGRATION

20.1.0

Table of Contents

1. Summary

The Stripe LINK Cartridge facilitates integration between a Commerce Cloud Storefront and Stripe Payment Services; including Stripe Elements, Sources, Webhooks, and Alternate Payment methods. Usage of Sources via Stripe.js, ability to create charges, and optional integration with Stripe's Relay service for embedded eCommerce solutions on social channels.

Contracting with Stripe is required for live, production deployment of the cartridge. Though the cartridge can be installed and tested using a freely available Stripe test account at https://dashbo ard.stripe.com. Please contact your Stripe Implementation Consultant for help with taking your Stripe account live.

The integration encompasses the deployment of several cartridges and modification of storefront code

2. Component Overview

Functional Overview

Stripe Elements and Sources

Stripe Elements modifies the default Commerce Cloud Credit Card collection and processing by using Stripe.js, a JavaScript library, to securely tokenize credit card data. Payments are then processed using the tokenized data, not the raw credit card information.

During checkout, the cartridge will create a source for any new cards or alternate payment methods entered by customers. This data is transformed into a Stripe Source. At the point of purchase, the stored, tokenized data is used to generate a Stripe Charge. Registered Customers can manage (add, delete) reusable payment methods in their storefront-connected Stripe Account for re-use in subsequent storefront purchases.

Use Cases

Stripe.js Sources

When customers enter credit card or other payment information on the storefront, the information is tokenized via Stripe.js in a client (browser)-to-Stripe interactions. Unmasked credit card data is therefore never sent to the Commerce Cloud servers.

Stripe Charges

System will create a Stripe Charge (authorize or capture, based on Business Manager configuration) from a successfully created and submitted Basket. All Stripe Charges are created against a Stripe payment Source.

AVS Auto-Fail Transactions

Site administrators can select a variety of AVS statuses for which an Order should be auto-failed. If the Stripe Charge returns any of the selected statuses for either address_line1_check or address_zip_check the Order will be auto-failed and the Stripe Charge reversed. Note that these settings can also be managed on the Stripe Dashboard.
Supported payment methods:

- Cards (Visa, Mastercard, American Express, Discover, Diners Club, JCB) Alipay
- The Payment Request Button Element gives you a single integration for Apple Pay, Google Pay, Microsoft Pay, and the browser standard Payment Request API.

Limitations, Constraints

Stripe offers a number of standard services that are not supported by the cartridge. These include support for Subscriptions, Plans, and Coupons. There aren't any known locale specific restrictions in the cartridge.

The included RELAY OCAPI configurations are included as examples only. A RELAY implementation will require additional configuration and testing along with the Stripe team.

For any locale specific restrictions, please refer to https://stripe.com/docs/js.

Compatibility

Available since Commerce Cloud Platform Release 16.8, SFRA version 4.4

The cartridge is available for installations on storefronts that support both Controller and SFRA SiteGenesis implemenations.

Privacy, Payment

No unmasked credit card data is stored within Commerce Cloud. The cartridge tokenizes all payment data via direct client-to-Stripe communications and obscures any sensitive credit card data before it arrives on the Commerce Cloud servers. Similarly, all credit card data that is retrieved by Commerce Cloud from the Stripe servers is also masked and/or tokenized.

3. Implementation Guide

Setup of Business Manager

The Stripe LINK Cartridge contains several cartridges that are required for full functionality. Additionally, Controller and SFRA support is broken out into two separate cartridges, thereby facilitating the installation and use of one or the other models.

Import all three cartridges into UX studio and associate them with a Server Connection.

Site Cartridge Assignment

1. *Navigate to Administration > Sites > Manage Sites*
2. *Click on the Site Name for the Storefront Site that will add Stripe Functionality*
3. *Select the "Settings" tab*
4. *For SFRA "app_stripe_sfra:int_stripe_sfra:int_stripe_core" needs to be added to the cartridge path*
5. *Repeat steps 2 – 4 for each Storefront Site where Stripe will be implemented*

Business Manager Cartridge Assignment

1. *Navigate to Administration > Sites > Manage Sites - Click on the Business Manager Site > "Manage the Business Manager site." Link*
2. *Add "int_stripe_core" to the Cartridges: path*

Metadata import

1. *Navigate to the metadata folder of the project and open the stripe_site_template folder.*
2. *Open the sites folder and edit the 'siteIDHere' folder to the site ID of the site you want.*
3. *Add a folder for each site you need stripe on.*
4. *Navigate to Administration > Site Development > Site Import & Export*
5. *Zip the stripe_site_template folder and import it.*

Building Stripe Styling

If necessary, update the path to your base SFRA installation in package.json file from the same root folder.

Normally you would have a top-level project folder, into which the repositories of SFRA base cartridge and all required plugins, libraries and any other LINK cartridges will be cloned. In case you have cloned the Stripe cartridge into that folder as well, the below change will not be required. Otherwise, update paths.base property in the package.json to contain a relative path to the local directory containing the Storefront Reference Architecture repository. Its default value will be as follows:

```
"paths": {
    "base": "../storefront-reference-architecture/cartridges/app_storefront_base/"
}
```

Once you are certain the correct path to SFRA cartidges is configured, run npm run compile:scss command from the root folder of Stripe repository.

Add New Payment Processors

There are two payment processors used in the Stripe cartridge. "STRIPE_CREDIT" is used for credit card handling while "STRIPE_APM" is used for the asynchronous payment model (Bank transfers, GiroPay, etc).

If using Stripe credit cards, Navigate to Merchant Tools > Ordering > Payment Processors and click the "New" button. In the new window set the ID attribute to value "STRIPE_CREDIT" and click "Apply".

If using APM methods, again, click the "New" button. In the new window set the ID attribute to value "STRIPE_APM" and click "Apply". This payment method is for the non-credit card (APM methods)

Update Payment Methods

Navigate to Merchant Tools > Ordering > Payment Methods, click on the CREDIT_CARD payment method and select the STRIPE_CREDIT payment processor in dropdown under the CREDIT_CARD Details section

If using APM payment methods and/or the Payment Request Button then enable the desired payment methods as well: The STRIPE_APM_METHODS will provide the ability to include all of the supported Stripe methods. See https://stripe.com/payments/payment-methods-guide

To utilize the Stripe Payment Request Button, enable the "STRIPE_PAYMENT_REQUEST_BTN" payment method. See https://stripe.com/docs/stripe-js/elements/payment-request-button

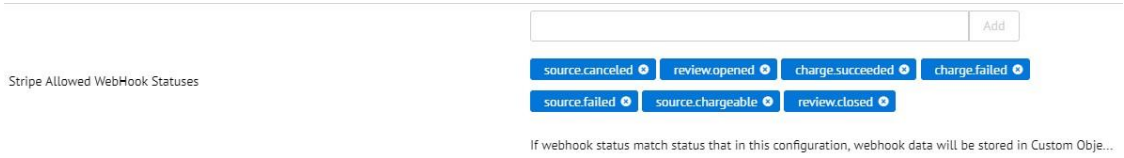
Configuration

Update the Merchant Tools > Site Preferences > Custom Site Preferences > Stripe Configurations with Site specific values.

1. Stripe Secret API Key a. Can be obtained through the Stripe Dashboard (https://dashboard.stripe.com/account/apikeys)
2. Stripe Publishable API Key a. Find along with Stripe Secret API Key
3. Is this SFRA installation. Set to yes if the current site is using the Storefront Reference Architecture (SFRA)
4. Capture Funds on Stripe Charge a. Default value: true (Yes) b. Set to false (No) to instead Authorize Stripe Charges
5. Stripe Card Element CSS Style a. Enter the CSS styling that the Card element button should inherit to fit within the overall storefront styles. Style Configuration for Stripe Elements e.g, {"base": {"fontFamily": "Arial, sans-serif","fontSize": "14px","color": "#C1C7CD"},"invalid": {"color": "red" } }
6. Stripe API URL - https://js.stripe.com/v3/
7. Stripe Payment Request Button Style a. For the payment request button, select the limited CSS styling that the button should display with. See https://stripe.com/docs/stripe-js/elements/payment-requestbutton#styling-the-element
8. ApplePay Verification String i. Enter the Apple verification string provided from the Stripe dashboard. ii. This is a one time enablement. The Stripe console will proxy the Apple Pay for Web verification String upon setup. This will need to be configured into the sandbox if the Payment Request Button will be used as a form of payment on the storefront.
9. Country Code (Stripe Payment Request Button) - Country Code e.g, US. This will be the default country code for the Payment Request Button. Customization may be needed on a multi country single site in order to dynamically pass the country code rather than the site preference (if needed). https://stripe.com/docs/stripe-js/elements/payment-requestbutton# create-payment-request-instance
10. Stripe Webhook Signing Secret i. Enter the webhook signing secret provided by the stripe dashboard. Stripe will sign webhook calls and pass a validation to SFCC. SFCC will validate the contents of the message via this key.
11. Stripe allowed Webhook Statuses i. Configure the allowed statuses for Webhooks to respond to.

Set to:

- review.opened
- review.closed
- charge.succeeded
- charge.failed
- source.canceled
- source.failed
- source.chargeable

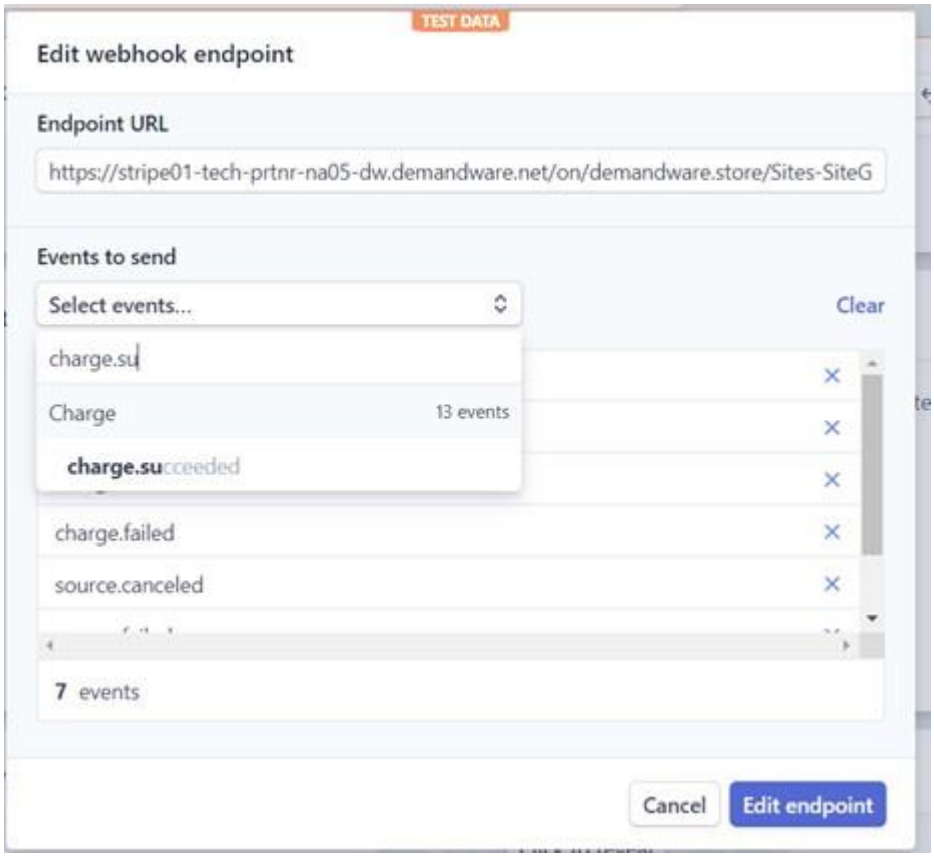


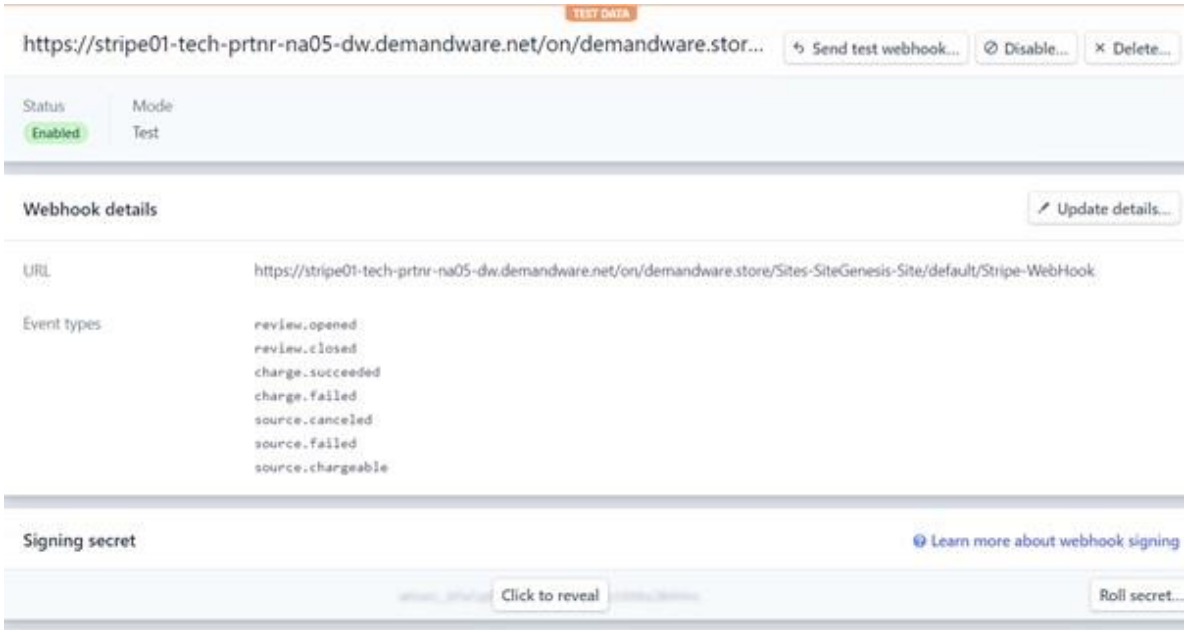
12. Allowed APM Methods a. Update this field, per site locale, to indicate which alternate payment methods are enabled for each locale. Enumeration of allowed Payment Methods from the Stripe API. See more here: https://stripe.com/docs/sources { "default": ["p24", "eps", "sepa_debit", "ideal", "sofort", "bitcoin", "alipay", "bancontact", "giropay"], "en_UK": ["p24", "eps"], "de_AT": ["sofort", "ideal"] }
13. Stripe Enabled – Enables or disables the cartridge

Stripe Dashboard

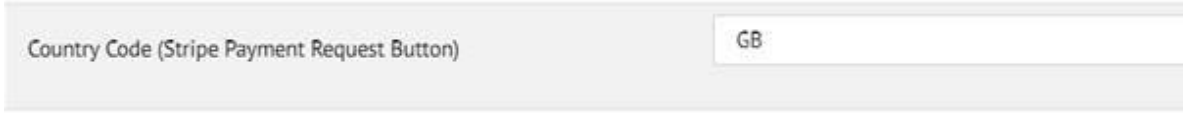
In the Stripe Dashboard (https://dashboard.stripe.com/test/webhooks) enable webhooks, point it to `Stripe-webHook` controller and subscribe to these events:

- review.opened
- review.closed
- charge.succeeded
- charge.failed
- source.canceled
- source.failed
- source.chargeable





Then copy the signing secret to the 'Stripe Webhook Signing Secret' preference.
Make sure that this value is set to your Stripe account country code:



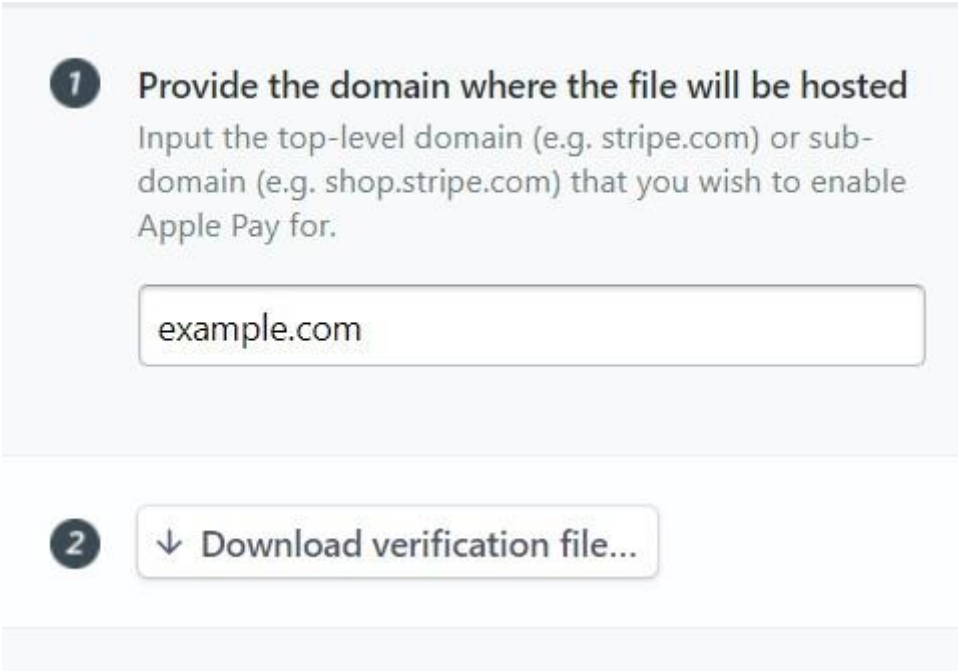
For ApplePay to work, the file RedirectURL.js must be changed with this code:

```
    if (URLRedirectMgr.getRedirectOrigin() === '/.well-known/apple-developer-merchantid-domain-association') { // Intercept the incoming path request
        res.render('stripe/util/apple');
        return next();
    }
```

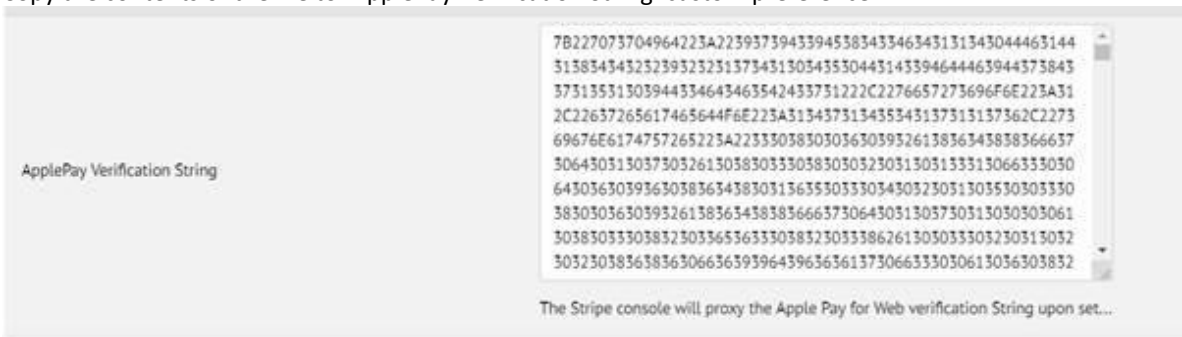
Then you then need to set an alias to one of the sites on the sandbox temporarily so the stripe dashboard can verify the domain. The alias needs to be something like this:

```
{
 "__version": "1",
    "settings": {
        "http-host": "your.sandbox.domain.demandware.net",
        "https-host": "your.sandbox.domain.demandware.net",
        "default":"true",
        "site-path": "/"
    },
    "your.sandbox.domain.demandware.net": [
        {
            "locale": "en_GB",
            "if-site-path": "/"
        }
    ]
}
```

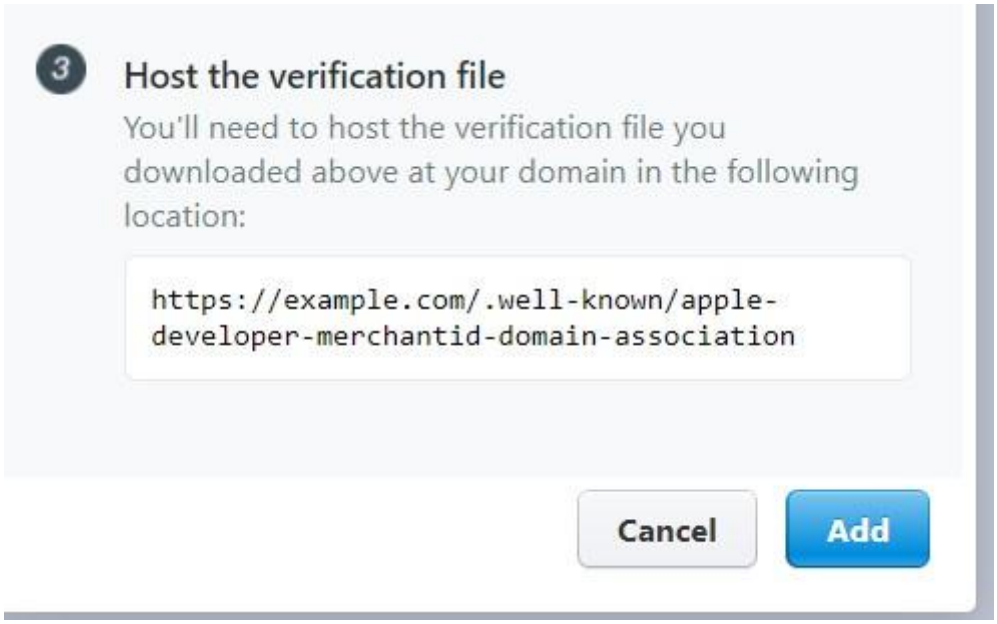
The locale value needs to be a locale that is not disabled.
Then go to https://dashboard.stripe.com/account/apple_pay and click on 'Add new domain' button. Enter the domain and download the verification file:



Copy the contents of the file to 'ApplePay Verification String' custom preference:



Then click on the 'Add' button:



Custom Code

The base LINK Cartridge code contains support for all credit cards supported by Stripe. Note that the list of allowed cards on the storefront is still limited by the Credit/Debit Cards list in Business Manager (Merchant Tools > Ordering > Payment Methods > Credit/Debit Cards).

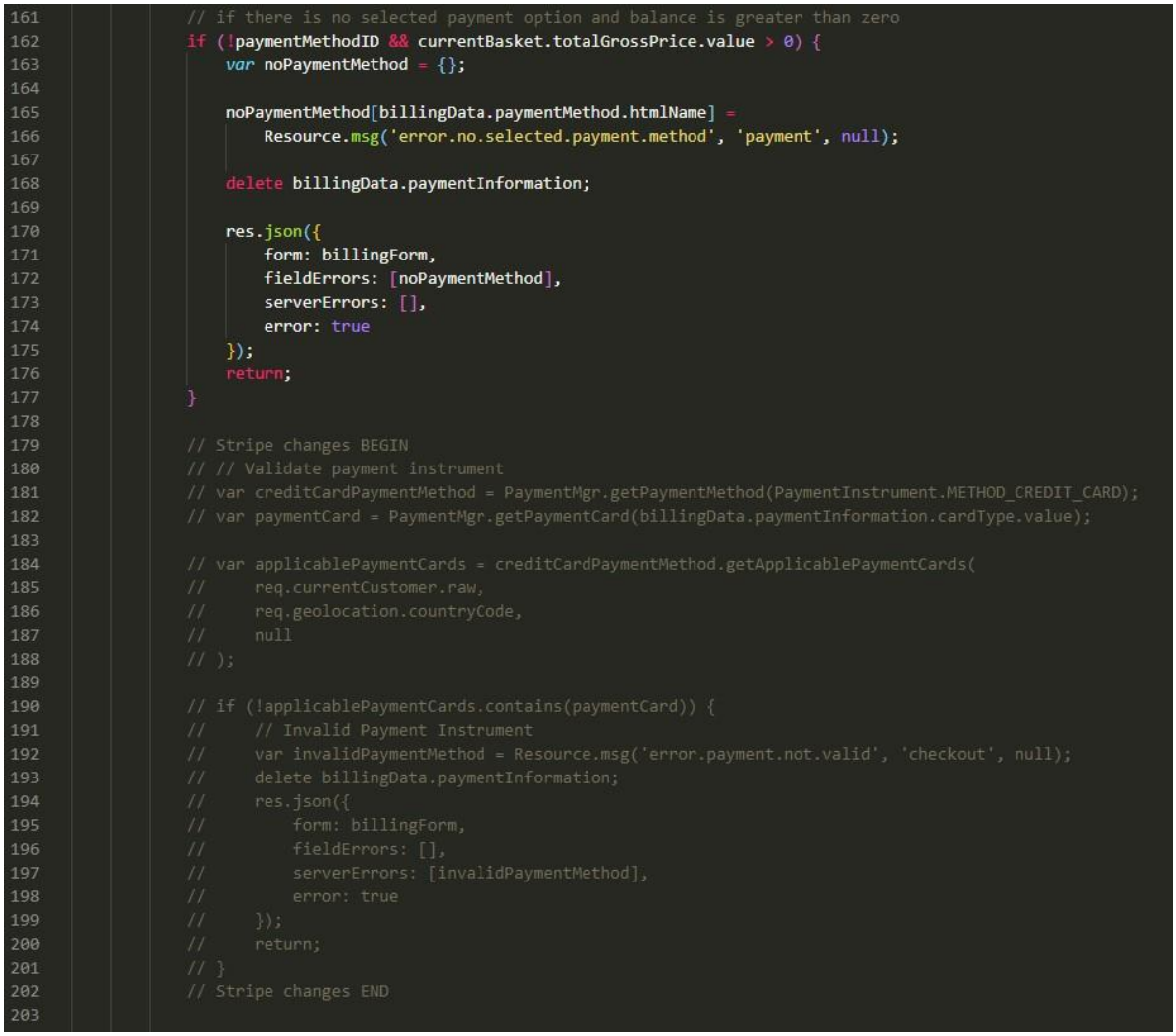
Make the following updates to the Storefront Code. Examples provided are based on SFRA version 4.4. Below are the customizations made to SFRA code.

There are a lot of controller endpoints that are appends instead of replaces. Those will not get covered as they should work without doing anything.

Controller updates are only required for replaced endpoints as you may have already replaced that endpoint in your integration. Simply use the changes that are made to the base cartridge and add them to your already replaced controller. If you haven't extended/replaced these endpoints you don't need to do anything.

Controller: CheckoutServices.js
app_stripe_sfra/cartridge/controllers/CheckoutServices.js

Remove the payment instrument validation in the 'SubmitPayment' endpoint



```
        // if there is no selected payment option and balance is greater than zero
        if (!paymentMethodID && currentBasket.totalGrossPrice.value > 0) {
            var noPaymentMethod = {};

            noPaymentMethod[billingData.paymentMethod.htmlName] =
                Resource.msg('error.no.selected.payment.method', 'payment', null);

            delete billingData.paymentInformation;

            res.json({
                form: billingForm,
                fieldErrors: [noPaymentMethod],
                serverErrors: [],
                error: true
            });
            return;
        }

        // Stripe changes BEGIN
        // // Validate payment instrument
        // var creditCardPaymentMethod = PaymentMgr.getPaymentMethod(PaymentInstrument.METHOD_CREDIT_CARD);
        // var paymentCard = PaymentMgr.getPaymentCard(billingData.paymentInformation.cardType.value);

        // var applicablePaymentCards = creditCardPaymentMethod.getApplicablePaymentCards(
        //     req.currentCustomer.raw,
        //     req.geolocation.countryCode,
        //     null
        // );

        // if (!applicablePaymentCards.contains(paymentCard)) {
        //     // Invalid Payment Instrument
        //     var invalidPaymentMethod = Resource.msg('error.payment.not.valid', 'checkout', null);
        //     delete billingData.paymentInformation;
        //     res.json({
        //         form: billingForm,
        //         fieldErrors: [],
        //         serverErrors: [invalidPaymentMethod],
        //         error: true
        //     });
        //     return;
        // }
        // Stripe changes END
```

Update 'PlaceOrder' as below:
Replace the order creation block:

```
    // Re-calculate the payments.
    var calculatedPaymentTransactionTotal=
COHelpers.calculatePaymentTransaction(currentBasket);
    if (calculatedPaymentTransactionTotalerror) {
        res.json({
            error: true,
            errorMessage: Resource.msg('error.technical', 'checkout', null)
        });
        return next();
    }
```

With this:

```
    const stripeCheckoutHelper =
require('int_stripe_core').getCheckoutHelper();
    var order = stripeCheckoutHelper.createOrder(currentBasket);
```

```
    // Re-calculate the payments.
    var calculatedPaymentTransactionTotal = COHelpers.calculatePaymentTransaction(currentBasket);
    if (calculatedPaymentTransactionTotal.error) {
        res.json({
            error: true,
            errorMessage: Resource.msg('error.technical', 'checkout', null)
        });
        return next();
    }

    // Stripe changes BEGIN
    const stripeCheckoutHelper = require('int_stripe_core').getCheckoutHelper();
    var order = stripeCheckoutHelper.createOrder(currentBasket);
    // Stripe changes END

    if (!order) {
        res.json({
            error: true,
            errorMessage: Resource.msg('error.technical', 'checkout', null)
        });
        return next();
    }
```

Replace the order placement down below too:

```
var placeOrderResult = COHelpers.placeOrder(order, raudDetectionStatus);
```

Replace everything after this line with:

```
var isAPMOrder = stripeCheckoutHelper.isAPMOrder(order);
    if (!isAPMOrder) {
        var stripePaymentInstrument =
stripeCheckoutHelper.getStripePaymentInstrument(order);

        if (stripePaymentInstrument &&
order.custom.stripeIsPaymentIntentInReview) {
            res.json({
                error: false,
                orderID: order.orderNo,
                orderToken: order.orderToken,
                continueUrl: URLUtils.url('Order-Confirm').toString()
            });

            return next();
        }
        // Places the order
        var placeOrderResult = COHelpers.placeOrder(order,
fraudDetectionStatus);
        if (placeOrderResult.error) {
            stripeCheckoutHelper.refundCharge(order);
            res.json({
                error: true,
                errorMessage: Resource.msg('error.technical', 'checkout', null)
            });
            return next();
        }

        COHelpers.sendConfirmationEmail(order, req.locale.id);

        // Reset usingMultiShip after successful Order placement
        req.session.privacyCache.set('usingMultiShipping', false);
```

```
        res.json({
            error: false,
            orderID: order.orderNo,
            orderToken: order.orderToken,
            continueUrl: URLUtils.url('Order-Confirm').toString()
        });

        return next();
    }
    res.json({
        error: false,
        orderID: order.orderNo,
        orderToken: order.orderToken,
        continueUrl: URLUtils.url('Order-Confirm').toString()
    });

    return next();
```

```
var fraudDetectionStatus = hooksHelper('app.fraud.detection', 'fraudDetection', currentBasket, require('*/cartridge/scripts/hooks/fraudDetection').fraudDetection);
if (fraudDetectionStatus.status === 'fail') {
    Transaction.wrap(function () { OrderMgr.failOrder(order); });

    // fraud detection failed
    req.session.privacyCache.set('fraudDetectionStatus', true);

    res.json({
        error: true,
        cartError: true,
        redirectUrl: URLUtils.url('Error-ErrorCode', 'err', fraudDetectionStatus.errorCode).toString(),
        errorMessage: Resource.msg('error.technical', 'checkout', null)
    });

    return next();
}

// Stripe changes BEGIN
var isAPMOrder = stripeCheckoutHelper.isAPMOrder(order);
if (!isAPMOrder) {
    var stripePaymentInstrument = stripeCheckoutHelper.getStripePaymentInstrument(order);

    if (stripePaymentInstrument && order.custom.stripeIsPaymentIntentInReview) {
        res.json({
            error: false,
            orderID: order.orderNo,
            orderToken: order.orderToken,
            continueUrl: URLUtils.url('Order-Confirm').toString()
        });

        return next();
    }
    // Places the order
    var placeOrderResult = COHelpers.placeOrder(order, fraudDetectionStatus);
    if (placeOrderResult.error) {
        stripeCheckoutHelper.refundCharge(order);
        res.json({
            error: true,
            errorMessage: Resource.msg('error.technical', 'checkout', null)
        });
        return next();
    }

    COHelpers.sendConfirmationEmail(order, req.locale.id);

    // Reset usingMultiShip after successful Order placement
    req.session.privacyCache.set('usingMultiShipping', false);

    // TODO: Exposing a direct route to an Order, without at least encoding the orderID
    //  is a serious PII violation.  It enables looking up every customers orders, one at a
    //  time.
    res.json({
        error: false,
        orderID: order.orderNo,
        orderToken: order.orderToken,
        continueUrl: URLUtils.url('Order-Confirm').toString()
    });

    return next();
}
```

Controller: PaymentInstruments.js
app_stripe_sfra/cartridge/controllers/PaymentInstruments.js

Replace the DeletePayment endpoint with this code

```
server.replace('DeletePayment', function (req, res, next) {
    var stripeHelper = require('int_stripe_core').getStripeHelper();
    var wallet = stripeHelper.getStripeWallet(customer);
    var UUID = req.querystring.UUID;
    wallet.removePaymentInstrument({ custom: { stripeId: UUID } });

    res.json({ UUID: UUID });
    next();
});
```

Controller: RedirectURL.js
app_stripe_sfra/cartridge/controllers/RedirectURL.js In the

function start add the following code:

```
    if (URLRedirectMgr.getRedirectOrigin() === '/.well-known/apple-developer-
merchantid-domain-association') { // Intercept the incoming path request
        res.render('stripe/util/apple');
        return next();
    }
```

External Interfaces

Stripe functionality relies heavily on external calls to the Stripe services. All external interfaces use the Service Framework to communicate with the Stripe API.

Stripe accounts are free to create and use. Most communications with Stripe services are logged and easily accessible in the Stripe Dashboard (http://dashboard.stripe.com). It is highly encouraged to use the Stripe Dashboard to monitor and test your integration.

The main configuration for integration of the Stripe services can be found under **Administration > Operations > Services**

There is a different service for each external call:

- stripe.http.addCard
- stripe.http.authorizePayment
- stripe.http.createCharge
- stripe.http.createCustomer
- stripe.http.deleteCard
- stripe.http.fetchCustomerCards
- stripe.http.fetchCustomerSources
- stripe.http.refundCharge
- stripe.http.retrieveCustomer
- stripe.http.service
- stripe.http.updateCard

All of these services use the same profile and the same credentials. The only thing that may be different is whether or not the communication log is enabled and the log name prefix. Here is the configuration of some of the services:

stripe.http.addCard

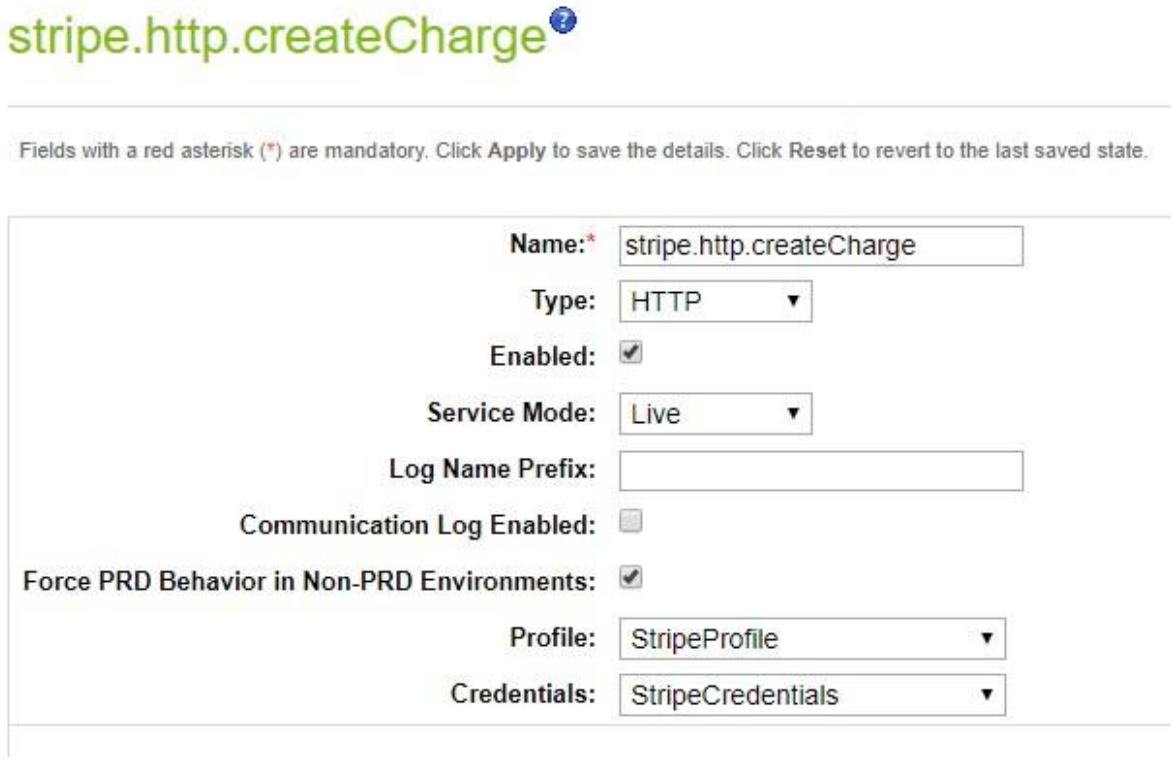
Fields with a red asterisk (*) are mandatory. Click **Apply** to save the details. Click **Reset** to revert to the last saved state.

Name:*	stripe.http.addCard
Type:	HTTP
Enabled:	☑
Service Mode:	Live
Log Name Prefix:	Stripe
Communication Log Enabled:	☑
Force PRD Behavior in Non-PRD Environments:	☐
Profile:	StripeProfile
Credentials:	StripeCredentials

stripe.http.authorizePayment

Fields with a red asterisk (*) are mandatory. Click **Apply** to save the details. Click **Reset** to revert to the last saved state.

Name:*	stripe.http.authorizePayment
Type:	HTTP
Enabled:	☑
Service Mode:	Live
Log Name Prefix:	Stripe
Communication Log Enabled:	☑
Force PRD Behavior in Non-PRD Environments:	☐
Profile:	StripeProfile
Credentials:	StripeCredentials



Firewall Requirements

No requirements

4. Testing

Please, find more details on the test case document in the same folder.

5. Operations, Maintenance

Data Storage

The Stripe LINK cartridge extends Commerce Cloud to store several data points.

Customer Profile: Stripe Customer ID, used to retrieve information about the customer's record in your Stripe account.

1. *stripeCustomerID(string) - Store Stripe customer ID*

Order/Basket Custom attributes

1. *stripePaymentIntentID(String) – Store payment intent ID.*
2. *stripeIsPaymentIntentInReview(Boolean) - Store payment intent in review*

Payment Transaction custom attributes

1. *stripeChargeId(string) - Store charge id*
2. *stripeChargeOutcomeData(text) - Store charge outcome data*
3. *stripeClientSecret(string) - Store client secret*
4. *stripeJsonData(text) - Store webhook JSON data*
5. *stripeOrderNumber(number) - Store order number*
6. *stripeSourceCanCharge(boolean) - Store if Stripe source can be charged*
7. *stripeSourceId(string) - Store Stripe source ID*

Payment Transaction custom attributes

1. *stripeChargeId(string) - Store charge ID*
2. *stripeCardID(string) - Store card ID*
3. *stripeCustomerID(string) - Store customer ID*
4. *stripeDefaultCard(boolean) - Store Stripe default card*
5. *stripeClientSecret(string) - Store client secret*
6. *stripePRUsed(boolean) - Store payment request button used*
7. *stripeSavePaymentInstrument(boolean) - Store save payment instrument*
8. *stripeSourceID(string) - Store Stripe source ID*

Custom Objects: In Business Manager, navigate to the Merchant Tools > Custom Objects > Custom Objects. Below custom object is there.

1. *StripeWebhookNotifications*

Custom Site Preferences: noted in detail above (section Configuration).

Availability

Please refer to the Stripe Service Level Agreement https://stripe.com/legal to determine specific up-times for the service. In case the service fails, there is no fail-over to allow transactions to proceed. Users will instead be provided with friendly error messaging.

Failover/Recovery Process

If the Stripe service is unavailable the user will not be able to checkout.

The service availability can be tracked in SFCC using the Service Status.

Support

For defects or recommendations on improvements, please contact Stripe Support (https://suppor t.stripe.com).

6. User Guide

Roles, Responsibilities

There are no recurring tasks required by the merchant. Once configurations and job schedules are set up, the functionality runs on demand.

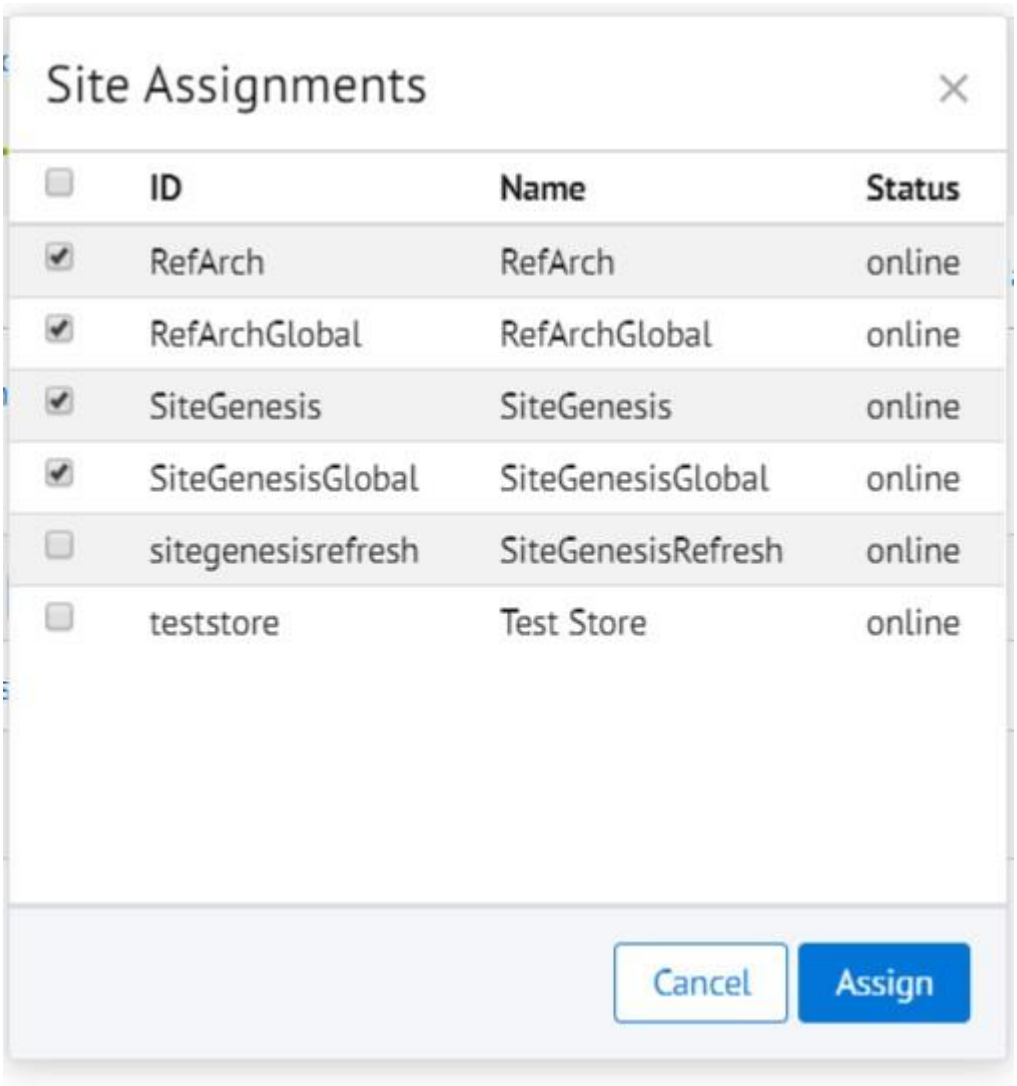
Business Manager

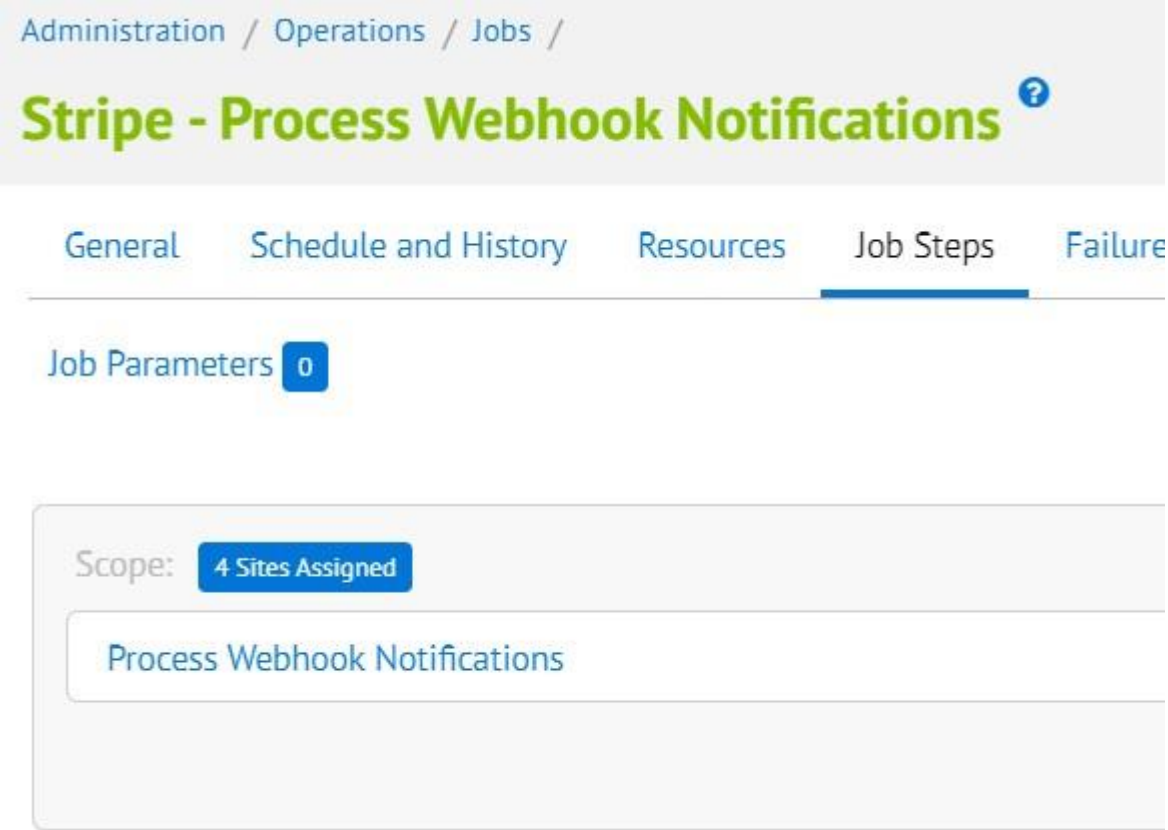
Business Manager settings and configuration notes are described in detail in the Configurations section.

There are 2 jobs coming with the cartridge:

- Stripe - Delete Custom Objects
- Stripe - Process Webhook Notifications

Enable the job “Stripe - Process Webhook Notifications” for the desired site:





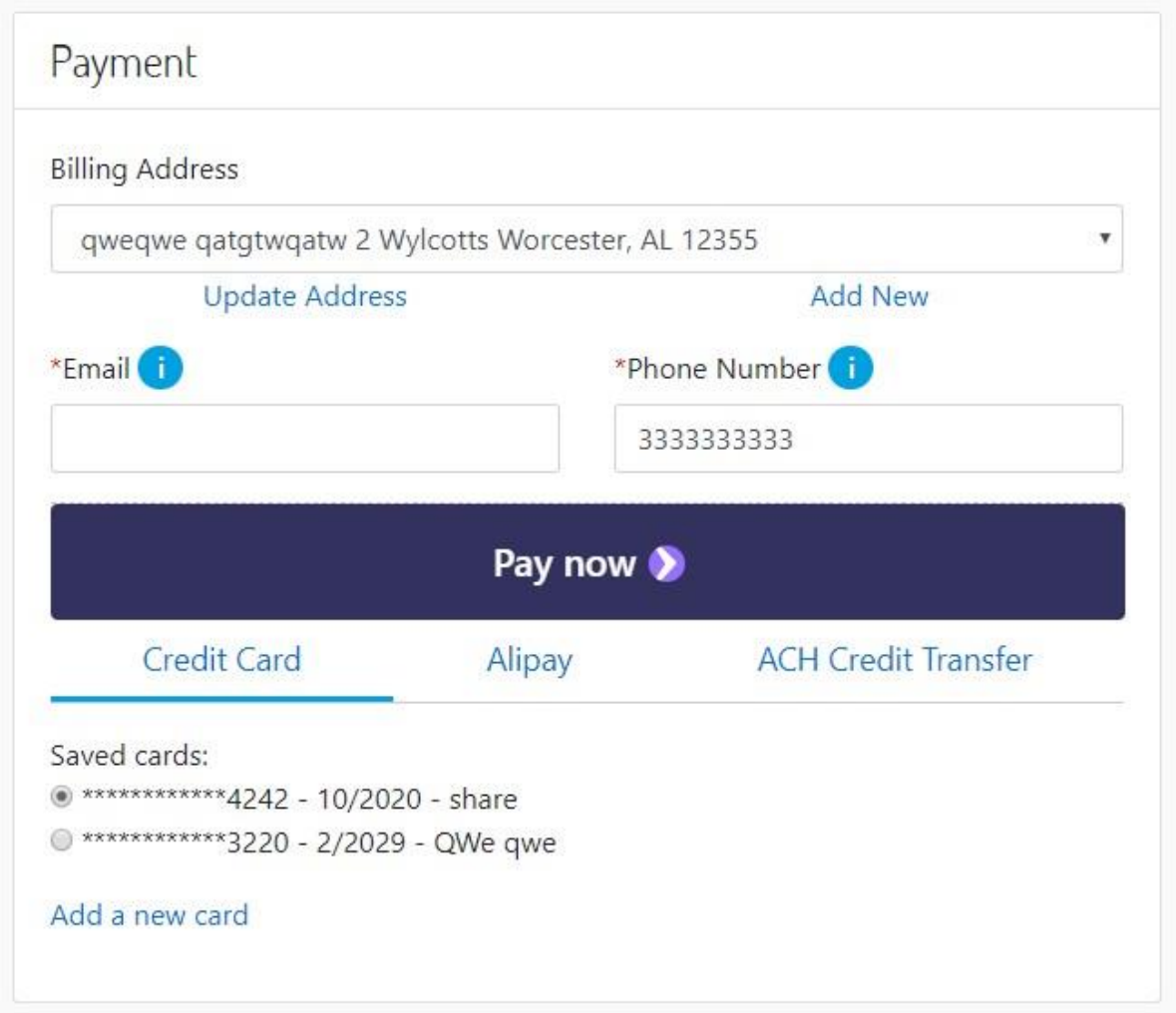
Storefront Functionality

Credit Card Tokenization

Stripe.js credit card tokenization requires the inclusion of JavaScript on the payment forms, both during Checkout > Billing as well as My Account > Saved Payment Instruments. Additionally, the credit card 'type' form fields are automatically detected and updated rather than requiring user selection.

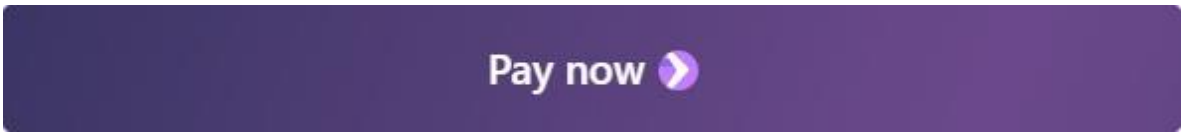
Saved Credit Cards

When an authenticated customer selects a saved credit card on the Checkout > Billing page, they will see a list of their Stripe-saved payment Sources as radio buttons rather than the default SiteGenesis select options.



Payment request button

When a customer has a saved address and credit card information in their browser they see the payment request button (Pay Now). The Payment Request Button Element gives you a single integration for Apple Pay, Google Pay, Microsoft Pay, and the browser standard Payment Request API.



Customers see the button above or an Apple Pay button, depending on what their device and browser combination supports. If neither option is available, they don't see the button. Supporting Apple Pay requires additional steps, but compatible devices automatically support browser-saved cards, Google Pay, and Microsoft Pay.

7. Known Issues

The LINK Cartridge has no known issues.

8. Release History

Version	Date	Changes
20.1.0	2020-02-01	Update documentation to match the new Salesforce template
18.1.0	2019-04-15	Update to use Stripe elements, sources, payment request button, webhooks and asynchronous payments
16.1.0	2019-07-30	Initial release