Library Documentation

QuickPayments.JS

MiCamp Solutions LLC

This library has a variety of customizations that will allow you a lot of flexibility but if the default behaviors work for you the library is very simple and requires only a couple of lines of html to use.

For developers the library is a component built with VueJS. The component works in HTML5, existing VueJS apps, Angular apps and React apps. We have examples of that later in the documentation. But, since the default behavior will work for most users we will first just cover the quick and basic setup and syntax.

The library has a lot of attributes, css classes, VueJS slots, and other ways to customize the library where you should seldom if ever need to make actual changes to the library code. In the case you do, you will need the full project, make your changes, do a production build and then use your version in place of the default build. In a VueJS app, you can just include the library files in your application if you don't want to do a separate build.

Basic Setup

If not merging this into an existing front-end build process, then this just requires a few lines in your HTML to get up and running.

1. CSS

In the head of your html bring in the CSS. This can be our standard CSS file and/or one of your own. To use the standard just add this line anywhere inside the **<head>** section of your html file.

With Bootstrap

Our library uses the class names also used by Bootstrap 4. If your app is already using Bootstrap then you are good. Just add the following somewhere after your Bootstrap line. Note, QuickPaymentsJS does **not** need the Bootstrap JS file so no need to include that if you don't want to.

```
<link rel="stylesheet" href="[our CDN path]/quickpayments.css">
```

No CSS Library

If not using Bootstrap, then you have a few options. If you are not already using a responsive, mobile-first CSS library then you should consider adding Bootstrap. In that case just add Bootstrap somewhere before the line above.

```
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css" crossorigin="anonymous">
```

Note that you don't need the Bootstrap JS file for QuickPaymentsJS.

Other CSS Libraries

If you are already using a responsive, mobile first CSS library, then later in the documentation, there is a list of the Bootstrap CSS class names that QuickPaymentsJS is using and you'll need to add those classes to your CSS or include our full CSS file instead of the slimmed down one shown above.

```
<link rel="stylesheet" href="[our CDN path]/bootstrap-quickpay.css">
```

2. Fonts

QuickPaymentsJS by default uses the free version of Font Awesome 5. You can override all the icons that are used if you want to use a different icon library. If you are already using FontAwesome then you are good to go. If not, you should really consider using Font Awesome to do so add this to your **<head>** also.

```
<link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.5.0/css/all.css" crossorigin="anonymous">
```

You can use either the Webfont (CSS) or SVG (javascript) version of FontAwesome.

3. Scripts

Somewhere in your html file (usually near the bottom of the file right before the closing **</body>** tag) you need to load both VueJS library and the QuickPaymentsJS library.

```
<script src="https://unpkg.com/vue"></script>

<script src="[our CDN path]/quickpayments.umd.min.js"></script>
```

4. Bootstrap QuickPaymentsJS

Somewhere after the scripts lines shown in step 2 (but before the closing **</body>** tag) add the following:

```
<script>
  new Vue({
    components: {
      payment: quickpayments.payment,
      paymentModal: quickpayments
    },
    data: {
    }
  }).$mount('#app')
</script>
```

You only need to include the elements that you are using. So, if you won't use the QuickPaymentsJS modal then you don't need to include the paymentModal line. Also, this allows you to override the element names if you wish. The above will use the standard names used throughout the documentation: **<payment>** and **<payment-modal>** but you can change those to whatever html compliant name you would like.

5. Add to your HTML

Finally, add to your HTML, in the exact place you want QuickPayments to appear, the element you wish to use.

```
<payment api-key="your api key from our web portal" post-url="URL to your server API"/>
```

See, the Basic Syntax section for what needs to go into the **api-key** and **post-url** attributes. Once that is done, you are done with all the front-end work. This will let you start taking payments with the default settings!

6. Install Server App

QuickPaymentsJS keeps all the customer's sensitive information off of your servers keeping you PCI compliant. To do this, QuickPaymentsJS sends the sensitive information directly to our servers and returns you a one-time-use, short-lived token that identifies that data on our servers. After, you receive that token, QuickPayments is ready to make the actual transaction. But, to do that securely, the transaction must come from your server (not from a browser) that will authenticate against our servers and send the token gained above, along with the remaining payment information.

The QuickPaymentsJS library handles most of this for you but, it does mean you need, at a minimum, a small app running on your servers. We have all the code you need to do the basic transactions, in a few programming languages. In many cases all you will need to do in that code is add your authentication login and password for our servers and then start the app up. If you want or need to do more with that data, you can use our code as a starting point. You will need the URL to your server and the path to that app. **This URL must be over HTTPS!**

Basic Syntax and Bindings

The basic syntax is the same for both **<payment>** and for **<payment-modal>**. None of these have default values. If either the **api-key** or **post-url** attributes are missing the component won't load and you'll see an error message instead. If you don't have the **:total-amount** attribute then the component will also show an error and won't be able to complete a transaction.

VueJS bind syntax is **v-bind:attribute=""** but the shortcut for that is **:attribute=""**. When an attribute starts with either 'v-bind:' or just ':' then this attribute is calculated by the VueJS library. This means for any of the attributes listed you can prepend a ':' to the attribute and instead pass in a javascript variable. When an attribute already starts with a ':' then QuickPaymentsJS expects a javascript variable of a certain type as listed in the description.

But to handle that binding, so it is "live", it requires that you register your variable with the Vue instance by listed it in the data section of the Vue object you created in the bootstrap step above. E.g. if you wanted to use **:total-amount="total"** where total is a variable.

```
<script>
   new Vue({
      components: {
         payment: quickpayments.payment
      },
      data: {
         total: a fixed value like 2.50 or your JS variable from your code
      }
   }).$mount('#app')
</script>
```

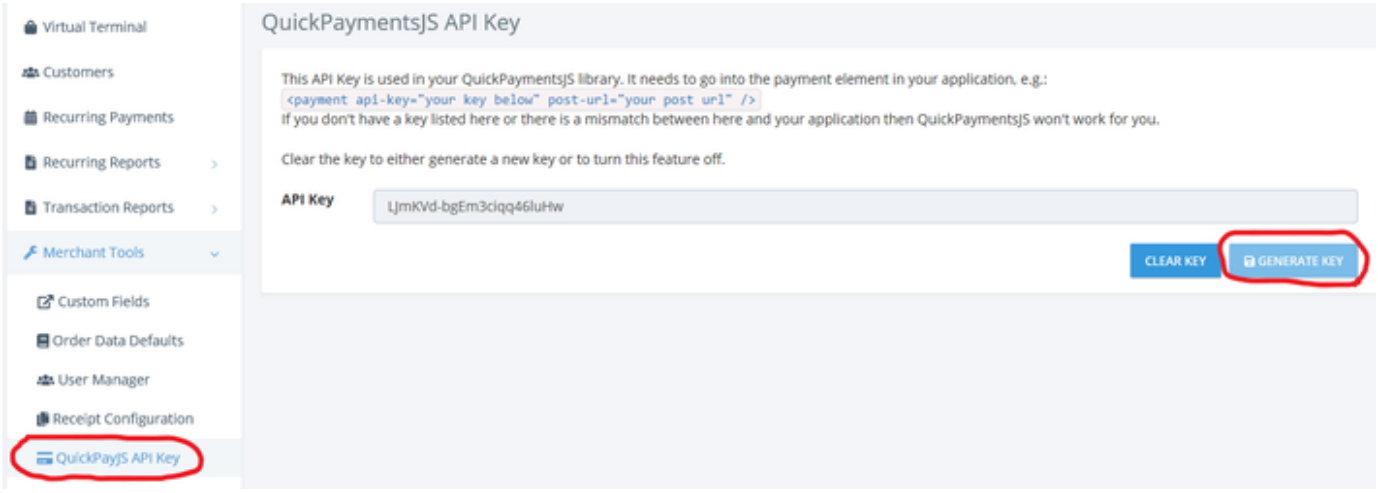
A note on bindings, if you bind an attribute then VueJS expects a variable! So, if you want to bind to an actual string then either don't using binding (the better option) or you need to quote the string: **:api-key=**"'a string'", compared to: **:api-key=**"aVariableName".

Required Settings

The only attributes required on the payment element are **api-key** and **post-url**. If you are actually accepting a payment and not just doing a tokenization request then the :**total-amount** attribute is also needed.

api-key=""

You can get your **api-key** from our portal, http://[link to the MPC UI], and log into your account. Under Merchant Tools menu there is a QuickPayJS API Key item. You may have to click on Merchant Tools to open the submenu. Click on the QuickPayJS API Key item and you'll see the page below. At first there will be no key there and you can click the Generate Key button to have the gateway generate a new api key for you. You can also replace your current key. To do so you need to first click Clear Key button, then you will be given the option to click Generate Key to receive a new key. It is best to select and copy the key directly from your browser and paste that directly in between the quotation marks for the **api-key** attribute in your html file.



post-url=""

QuickPaymentsJS keeps all the customer's sensitive information off of your servers keeping you PCI compliant. To do this, QuickPaymentsJS sends the sensitive information directly to our servers and returns you a one-time-use, short-lived token that identifies that links to that information on our servers. After, you receive that token, you're now ready to make a transaction. But, to do that securely, the transaction must come from your server (not from a browser) that will authenticate against our servers and send the token gained above, along with the remaining payment information.

The QuickPaymentsJS library handles most of this for you but, it does mean you need, at a minimum, a small servlet app running on your servers. We have all the code you need to do the basic transactions, in a few languages. In many cases all you will need to do in that code is add your login and password and then start running it. If you want or need to do more with that data, you can use our code as a starting point.

The **post-url** attribute needs to point to the URL where that server code is running. **This needs to be over HTTPS!**

qp-url=""

This is the URL to the QuickPayments tokens API. This URL will use your api-key and all the PCI data to send back a private token that will be sent to the post-url to tie the remaining payment data will the PCI data now stored on the gateway servers. So, qp-url is the URL to the gateway and post-url is the URL to your servers. This should be set to https://[link to server]/api/quickpayments/qp-tokens.

:total-amount=""

This is the amount of the transaction. This can be a number or a javascript variable that is a javascript Number type. If you are using this form just to tokenize the payment information and not to do an actual transaction then you can set this to 0.

Optional Attributes

The following are all optional and allow you to customize QuickPaymentsJS. The following list also shows the default setting that is used if you don't include the attribute. The following settings are used and are the same for both the **<payment>** and for the **<payment-modal>** components.

General Settings

icon-color="inherit"

This sets the icon-color of all the prepended and appended icons to the text input elements in QuickPaymentsJS. The value can be any valid CSS color, e.g. "silver", "#FF0000", etc.

card-btn-title="Card"

The sets the title of the card button on the payment page. Only used if you accept both payment types.

check-btn-title="Check"

The sets the title of the check button on the payment page. Only used if you accept both payment types.

pay-btn-title="Pay Now"

This sets title of the pay button. You can include the placeholder '%$' in this string. If you do then that placeholder will be replaced with the amount being paid, i.e. the value of **:total-amount** attribute. E.g.: **"Pay %$ Now"** would be displayed as "Pay $2.50 Now" if the total amount was 2.50.

next-btn-title="Next"

This sets title of the next button when there are multiple pages on the payment element.

prev-btn-title="Prev"

This sets title of the previous button when there are multiple pages on the payment element.

add-page=""

This allows you to create an additional page on the payment component to collect more information. If this is set it both enables the extra page and sets the title of that page to show to the customer. This attribute is only checked during the component being mounted into the DOM. So, even though you can bind this attribute with **:add-page** it won't live update after being put into the DOM.

pay-page-title="Payment"

The title to use for the Payment page.

bill-page-title="Billing Address"

The title to use for the Billing Address page.

ship-page-title="Shipping Address"

The title to use for the Shipping Address page.

gateway-custom-fields=""

If you have setup custom fields in the gateway web app then you can add those fields to this component (see Custom Input & Custom Field section below). As you can also see in the Server Code section, you have to manually handle additional fields that you add, the exception are these fields. This is a space-delimited list of field names. **Warning:** These names **must exactly** match the name of the fields you setup in the gateway ***and*** must exactly match the name attribute on the corresponding **input** (or **qp-input**) element.

Credit/Debit Card Settings

accept-cards="true"

This sets if you want to accept credit card payments. This can be true, false or a javascript variable that is the javascript boolean true or boolean false. If you are accepting both cards and checks then a pair of toggle buttons will appear letting the customer choose which to use.

capture-name="true"

This sets if you want to accept the card holder's name. For ACH payments a name must always be captured so there is no attribute for that. This can be true, false or a javascript variable that is the javascript boolean true or boolean false.

capture-zip="false"

This sets if you want to capture the customer's billing zip code. This can be true, false or a javascript variable that is the javascript boolean true or boolean false. If the **:billing-address** attribute is true then this setting is ignored as the billing zip will always be captured along with the full billing address.

cc-icon="far fa-credit-card"

This sets the icon to use for the credit card input element.

card-placeholder="Card number"

This sets the placeholder text for the card number input element.

name-icon="fas fa-user"

This sets the icon to use for the name input element for both credit cards and check.

name-placeholder="Name on card"

This sets the placeholder text for the card holder input element.

cvv-icon="fas fa-lock"

This sets the icon to use for the CVV input element.

cvv-placeholder="CVV"

This sets the placeholder text for the card's CVV input element.

expiry-icon="far fa-calendar-alt"

This sets the icon to use for the card expiration input element.

expiry-placeholder="MM / YY"

This sets the placeholder text for the card expiration input element.

cc-zip-icon="far fa-address-card"

This sets the icon to use for the card's billing zip code input element.

cc-zip-placeholder="Postal code"

This sets the placeholder text for the card's billing zip code input element.

visa-icon="fab fa-cc-visa fa-lg"

This sets the icon appended to the card's number input element when the card is a VISA.

mc-icon="fab fa-cc-mastercard fa-lg"

This sets the icon appended to the card's number input element when the card is a MasterCard.

amex-icon="fab fa-cc-amex fa-lg"

This sets the icon appended to the card's number input element when the card is a American Express.

discover-icon="fab fa-cc-discover fa-lg"

This sets the icon appended to the card's number input element when the card is a Discover.

diners-icon="fab fa-cc-diners-club fa-lg"

This sets the icon appended to the card's number input element when the card is a Diner's.

jcb-icon="fab fa-cc-jcb fa-lg"

This sets the icon appended to the card's number input element when the card is a JCB.

unknown-card-icon="fab fa-credit-card fa-lg"

This sets the icon appended to the card's number input element when the card is unknown.

Check (ACH) Settings

accept-checks="false"

This sets if you want to accept ACH payments. This can be true, false or a javascript variable that is the javascript boolean true or boolean false. If you are accepting both cards and checks then a pair of toggle buttons will appear letting the customer choose which to use.

routing-icon="fas fa-money-check"

This sets the icon to use for the bank routing input element.

routing-placeholder="Bank routing number"

This sets the placeholder text for the bank routing input element.

account-icon="fas fa-hashtag"

This sets the icon to use for the bank account input element.

account-placeholder="Account number"

This sets the placeholder text for the bank account input element.

name-icon="fas fa-user"

This sets the icon to use for the name input element for both credit cards and check.

check-name-placeholder="Name on check"

This sets the placeholder text for the check's name input element.

Address Settings

billing-address="false"

This sets if you want to capture the customer's billing address. There are a couple of options here:

- A javascript boolean false, the string 'false', the number 0, or the string '0' to not capture a billing address.
- A boolean true, the string 'true', the number 1, or the string '1' to capture the billing address but it is not required for the customer to fill out. If the **:capture-zip** attribute is also set to true then the billing zip is required even though the rest of the fields are not.
- A string 'req', the string 'required' or the number 2, to capture the billing address and it is required for the customer to fill it out. In this case the **:capture-zip** attribute is ignored as the billing zip will be captured here.

This attribute is only checked during the component being mounted into the DOM. So, even though you can bind this attribute with **:billing-address** it won't live update after being put into the DOM.

shipping-address="false"

This sets if you want to capture the customer's shipping address. There are a couple of options here:

- A javascript boolean false, the string 'false', the number 0, or the string '0' to not capture a billing address.
- A boolean true, the string 'true', the number 1, or the string '1' to capture the billing address but it is not required for the customer to fill out.
- A string 'req', the string 'required' or the number 2, to capture the billing address and it is required for the customer to fill it out.

This attribute is only checked during the component being mounted into the DOM. So, even though you can bind this attribute with **:shipping-address** it won't live update after being put into the DOM.

address-placeholder="Street address"

This sets the placeholder text for the billing and shipping street address input element.

address2-placeholder="Street address"

This sets the placeholder text for the billing's and shipping's optional 2nd line street address input element.

city-placeholder="City"

This sets the placeholder text for the billing and shipping city input element.

zip-icon="far fa-address-card"

This sets the icon to use for the billing and shipping zip code input element.

zip-placeholder="Postal code"

This sets the placeholder text for the billing and shipping zip code input element.

Results Customization

We have a slot system that will allow you to add elements and pages to the component, see Custom Input & Custom Fields section for more information, and, in addition, allows you to customize the look of end user error messages and customize the look of the success screen shown upon a successful payment.

test=""

This attribute is used for your development testing. It allows you to check your templates / layout for both a fake gateway/network error and for the success display. If you set this to "error" then the component will immediately act like it received a network or gateway error during the initial loading of your page. Setting this to "success" will have the component immediately act like it just received an approved transaction result from the gateway.

If set to "log", then the component will print to the console the data being sent and received to servers, don't leave this in when you go to production!

If set to "demo", this will log data to the console and fake the calls to the servers returning a success message. If set to "demo-fail", this will log data to the console and fake the calls to the servers returning a failed transaction due to card being declined message.

Successful Transaction Result

To customize the success display add the following slot to the component.

```
<div slot="transaction-success">
  [Your markup]
</div>
```

If you don't include this slot then the default markup is used:

```
<h3 class="text-success"><i class="far fa-check-circle"></i> Success!</h3>
```

Error Display

To customize the how error messages are displayed add the following slot to the component. This slot requires an additional attribute, **slot-scope**, in order for you to gain access to the actual error message.

```
<div slot="qp-error" slot-scope="qpError">
  [Your markup]
</div>
```

The "qpError" for the slot-scope attribute is a javascript variable name for a javascript object, so you can rename it but it must follow the rules for variable names. This object will contain one parameter, **error**, that is a string that is the actual error message. So to display the message you will need to add this somewhere in your markup: **{{qpError.error}}**.

If you don't include this slot then the default markup is used:

```
<div class="alert alert-danger">
  <span>{{qpError.error}}</span>
</div>
```

This error slot only effects the errors that come from your server, the gateway or from other network errors. It isn't used to display component mounting errors - these are the errors that happen if the component itself hasn't been setup. Mounting errors are errors only you should see during setup and so aren't end user errors.

Payment-Modal Attributes

In addition to all the other attributes discussed above for the payment component the **payment-modal** component can take a few more attributes.

btn-class="btn btn-primary"

This CSS class(es) to set on the button that is used to open the modal dialog.

btn-title="Buy Now"

The title or caption on the button that is used to open the modal dialog.

title="Checkout"

The title or header of the modal dialog. The title is in a h5 element.

title-class="modal-title"

The CSS class(es) to set on the title's h5 element.

size="lg"

The size of the modal – this needs to be part of your (or your framework's) CSS markup. This will be appended to **"modal-"** to give the full CSS name: **"modal-lg"**, **"modal-sm"**, etc. This is added to the dialog's div element along with the CSS **"modal-dialog"**.

centered="false"

This takes a Javascript Boolean, if true then the CSS class **"modal-dialog-centered"** is added to the dialog's div element.

no-fade="false"

This takes a Javascript Boolean, this affects if the CSS class **"fade"** is added to the modal's **div** element and the modal backdrop's **div** element.

modal-class=""

This CSS class(es) are added the modal's **div** element in addition to the **"modal"** class.

header-bg-variant=""

header-text-variant=""

header-border-variant=""

This is the background, text and border color variant to use. This is designed to work with Bootstrap's variants: "success", "primary", "warning", etc. So, if you set each of these to **"primary"** then the body of your modal will gain the "bg-primary", "text-primary" and "border-primary" classes.

header-class=""

This CSS class(es) are added the modal header's **div** element in addition to the **"modal-header"** class and any variants from the previous variant attributes.

hide-header="false"

This takes a Javascript Boolean, if true then the header of the modal is hidden.

hide-header-close="false"

This takes a Javascript Boolean, if true then the close "x" button of the modal is hidden.

header-close-label="Close"

The string that is put on the header's "x" close button's aria-label attribute.

body-bg-variant=""

body-text-variant=""

This is the background and text color variant to use. This is designed to work with Bootstrap's variants: "success", "primary", "warning", etc. So, if you set each of these to **"primary"** then the body of your modal will gain the "bg-primary" and "text-primary" classes.

body-class=""

This CSS class(es) are added the modal body's **div** element in addition to the **"modal-body"** class and any variants from the previous variant attributes.

footer-bg-variant=""

footer-text-variant=""

footer-border-variant=""

This is the background, text and border color variant to use. This is designed to work with Bootstrap's variants: "success", "primary", "warning", etc. So, if you set each of these to **"primary"** then the footer of your modal will gain the "bg-primary", "text-primary" and "border-primary" classes.

footer-class=""

This CSS class(es) are added the modal header's **div** element in addition to the **"modal-footer"** class and any variants from the previous variant attributes.

hide-footer="false"

This takes a Javascript Boolean, if true then the footer of the modal is hidden.

return-focus=""

This is the element or id of the element that should gain the focus when the modal is closed.

no-close-on-backdrop="false"

This takes a Javascript Boolean, if true then clicking on the backdrop doesn't close the modal.

no-close-on-esc="false"

This takes a Javascript Boolean, if true then using the escape key doesn't close the modal.

no-enforce-focus="false"

This takes a Javascript Boolean, if true then the modal won't be forced to be focused.

lazy="false"

This takes a Javascript Boolean, if true then the modal is put in the DOM lazily.

In addition, to all the above, the modal footer is empty, but there is a slot option to allow you to customize the modal footer.

```
<div slot="modal-footer">
  [Your markup]
</div>
```

Custom Input & Custom Fields

You can add your own fields and inputs into the QuickPaymentsJS form. The data from your custom inputs are sent to your **post-url** endpoint, the parameter in that object will be the same as what you set the standard HTML **name** attribute to.

We have a slot system that will allow you to add elements to any of the three possible pages of the QuickPaymentJS component. In addition, you can add one more page that is only made up of your markup.

QuickPaymentJS also includes a custom element, **qp-input** that is used internally. You also have the ability to use this element which will give you a lot of validation for "free" and it will automatically handle the error updating of the form for you. This element can also handle prepending and appending icons to the input element. See the section in the documentation for how to use that element.

For most custom fields you add here, you will need to also handle on the server side also. There is a small list you can add that will be handled automatically by our default NodeJS server code if you are using that, any others, you will need to manually enter into the server code. See the Server App section below for more information.

Payment Page

Unlike the other "pages" in the component, the payment page is made up of a possible two sub-pages: a Card subpage and the Check subpage.

To add custom elements to the Card subpage, add your markup to a div or template element with the **slot** attribute set to add-to-card.

```
<div slot="add-to-card">
  [Your markup]
</div>
```

To add custom elements to the Check subpage, add your markup to a div or template element with the **slot** attribute set to add-to-check.

```
<div slot="add-to-check">
  [Your markup]
</div>
```

Billing Page

This page is only shown if you set the **:billing-address** equal to true. When shown you can add custom elements to it by adding your markup to a div or template element with the **slot** attribute set to add-to-billing.

```
<div slot="add-to-billing">
  [Your markup]
</div>
```

Shipping Page

This page is only shown if you set the **:shipping-address** equal to true. When shown you can add custom elements to it by adding your markup to a div or template element with the **slot** attribute set to add-to-shipping.

```
<div slot="add-to-shipping">
  [Your markup]
</div>
```

Additional / Custom Page

This page is only shown if you set the **:add-page** equal to a valid string. The string you set here is what will be used as the title of this page. When shown you can add custom elements to it by adding your markup to a div or template element with the **slot** attribute set to additional-page.

```
<div slot="additional-page">
  [Your markup]
</div>
```

Error Handling

If you are using the built in **qp-input** element you won't need to worry about this, but if you are creating your own elements that also need validation, then you must let the QuickPaymentsJS library know when the validation on that object changes so it knows when it can allow the customer to move on to the next step.

To let QuickPaymentsJS know about an error or no error, you need to call the following after every validation you do.

```
[vueInstance].$root.setQpError(elementName, errorMessage, pageKey, forTransactionType);
```

A description of each of the arguments. These are vital to get correct if you want the error handling and form validation to be handled correctly.

- **elementName** is merely the **name** used on the element.
- **errorMessage** should either be description of the error (if there is an error) or **null** if no error.
- **pageKey** is the name of the QuickPaymentsJS page this is on. Your choices are: 'pay', 'bill', 'ship', 'add' for the payments page, billing address page, shipping address page and the custom additional page respectively.
- **forTransactionType** is only used for elements that are on the 'pay' page. It lists if the element is on the 'card' or 'check' subpage.

The **vueInstance** is the variable that is holding the Vue Instance that QuickPayments has been bootstrapped into. You would set that in the bootstrap step by assigning the new Vue() to a variable. If your app is in VueJS and you've pulled this library straight into your app then when inside a Vue function you can use **this**.

QP-Input

If you wish to use the built-in input element you will need to add that to the bootstrap script:

```
<script>
   new Vue({
      components: {
         payment: quickpayments.payment,
         qpInput: quickpayments.quickPaymentsInput,
         paymentModal: quickpayments
    }
   }).$mount('#app')
</script>
```

This input handles most of the validation for you, through attributes, allows you to additionally add a validation callback and automatically handles the error handling for you. The basic setup is the same as a normal HTML **input** element. And if you want the value to be sent to you then make sure to include a **name** attribute!

QP-Input Attributes

prepend-icon=""

This sets the icon to prepend to the input element. If not set, then none is shown.

append-icon=""

This sets the icon to append to the input element. If not set, then none is shown.

icon-color="inherit"

This sets the color of any icon prepended or appended to the input element. This can take any valid css color.

input-class=""

This allows you to add an additional css class to the internet HTML **input** element itself. This will be added to the always used classes of 'form-control' and either 'is-invalid' or 'is-valid'.

formatter=""

This must be a javascript function! If this attribute exists, then it is called after every **keyup** event to allow you a chance to override the formatting as the customer types. The function will receive a single argument which is the input's current value as a string. See **mask** below for how these two are related.

For some examples of this, there are two default inputs that use this.

1. The card expiration input uses this to add prepend a zero to the beginning of the string when the customer types a number greater than 1 for the first digit of the month:

```
formatExpiry(val) {
   if (val.length === 1 && parseInt(val) > 1) {
      val = '0' + val;
   }
   return val;
}
```

1. The card number input, on the other hand, doesn't use this to change the formatting but instead to update the appended credit card icon based on the current customer input.

mask=""

If this exists, then **qp-input** will check the current text value with this mask string on every **keydown** event. The mask sets the overall formatting for the input element and **only works for inputs that are numbers**. For examples of this, the card number. CVV and card expiration inputs use this. The mask string should have an 'X' wherever you want a number. This shows where in the string there should be a number, or space or slash, etc. And it also limits the input to this size. Examples of some of the masks used internally:

American Express card: 'XXXX XXXXXX XXXXX'

Other credit card: 'XXXX XXXX XXXX XXXX'

Expiry Date: 'XX / XX'

American Express card CVV: 'XXXX'

Other CVV: 'XXX'

maxlength="256"

Max length allowed for the input string. This isn't needed when using a mask.

minlength="0"

Min length allowed for the input string. This isn't needed when using a mask. If you set a **minlength** and don't mark this input **required** then **minlength** is only used once the length is greater than 0. In other words, **minlength** doesn't make this input required.

init-value=""

If set, then this is the value that will be set in the input when the component loads.

name=""

If set, then this is the value that will be set in the input when the component loads. **This is required if you actually want access to the data typed by the customer!**

numbers-only="false"

If set, then this input will only allow numbers. This can take a string or a boolean (if using **:numbers-only** version). Since this can take a string then you can use true, false, 1, or 0.

placeholder=""

Placeholder to use when the input is empty.

required="false"

If set then this input is required; in other words, there must be some input to continue on. This can take a string or a boolean (if using **:required** version). Since this can take a string then you can use true, false, 1, or 0.

placeholder=""

The title to set on the internal HTML **input** element.

type="text"

The input type to set on the internal HTML **input** element. So, any HTML5 option is valid here as it is just passed directly to the HTML **input** elem ent.

verify=""

This must be a javascript function! If this exists then it is called after being placed into the DOM, after every paste into the **input** element and after every **keyup** event. This allows you to do extra validation on the input. This is only called if all the other validations are good so far (**required** , **maxlength**, etc).

The format of the callback function is **functionName(event, currentStringValue)**. You **must return** either a **null** if valid or a string describing the error if invalid.

CSS Classes

For all the default elements in QuickPaymentsJS, there are specific CSS class wrappers giving you the ability to customize each element. All our elements and any elements you add using **<qp-input>** have some standard wrappers in addition to the individual wrapper.

Here is an example of how **<qp-input>** renders. The first div will also include any classes you set with the **input-class** attribute on the **<qp-input>** element.

```
<div class="input-group quickpay-input-wrapper [your input classes]">
   <div class="input-group-prepend quickpay-input-prepend">
      <span class="input-group-text"><i class="far fa-credit-card" …></i></span>
   </div>
   <input class="inputClass form-control" …>
   <div class="input-group-append quickpay-input-append">
      <span class="input-group-text"><i class="far fa-credit-card" …></i></span>
   </div>
</div>
```

For our default elements, there are also unique **input-class** classes for each element type.

CSS Class	Is on the div wrapper around the …
card-number	card number input
card-holder	card holder's name input
expiry	card expiration input
cvv	card CVV input
zip	card zip input
routing-number	check routing number input
account-number	account number input
check-name	name on check input
bill-address	billing street address input
bill-address2	2nd line of billing street address input
bill-city	billing city input
bill-state-select	billing state select element
bill-zip	billing zip input
ship-address	shipping street address input

ship-address2	2nd line of shipping street address input
ship-city	shipping city input
ship-state-select	shipping state select element
ship-zip	shipping zip input

Server Code

You will need at least a small app running on your server that QuickPaymentsJS can connect with to finish a transaction. We have a NodeJS app you can download and use or use as a template. You will need to edit the config.js file with your username/Merchant Key and password before running. In this code you will be able to handle custom fields and even record portions of the data that is being sent to the gateway and comes back from the gateway. Note that none of this data is PCI data.

Most server providers have NodeJS running on their servers by default, but almost all can add that for you if you ask, or if you have permissions, you can add it yourself, see https://nodejs.org/en/ for downloads.

The server code will automatically handle the following custom fields for you, if you add them to the component. In other words, the name attribute on your input field must exactly match one of the following:

- invoiceNumber
- orderNumber
- taxAmount
- tipAmount
- shippingAmount
- convenienceAmount
- surchargeAmount
- cashbackAmount

Other custom fields you will have to handle yourself. Many of these you will have to also log or put in your own database. The exception to this is any custom fields you have setup on the gateway web app those custom fields will be stored in the gateway database for you. Now to get that information to the gateway app, you must either manually add these to the customFields parameter in the server or the easier way is to merely add those custom field names to the **gateway-custom-fields** attribute. For those fields the Node app will automatically add those for you.

The **gateway-custom-field**s attribute is a space-delimited list of field names. **Warning:** These names **must exactly** match the name of the fields you setup in the gateway ***and*** must exactly match the name attribute on the corresponding **input** (or **qp-input**) element.

Note, you should really have someone setup a shell script or process manager that can watch the server app and restart it in case of an error or server restart. There are a few options out there for node process managers.

Including QuickPaymentsJS in Existing App Builds

You can add QuickPaymentsJS to a completed build as shown above, manually into the built files. But, that's not the best way to handle that when building full front-end apps in other frameworks. Adding this to a VueJS app is the same as adding any other VueJS component. In the case of VueJs you can choose to import the non-minified version if you wish to let it build as part of your project bundle. Here's how to add QuickPayments library to Angular and React apps.

Angular

1. Download the minified QuickPayments library and minified VueJS library.
2. In the Angular component that you want to use QuickPayments, load those two files at the top of the file.

```
import Vue from '[path to]/vue.js';
import QuickPayments from '[path to]/quickpayments.umd.min.js';
```

1. In the ngOnInit() hook bootstrap the QuickPayments component.

```
ngOnInit() {
   new Vue({
      components: {
         payment: QuickPayments.payment
      }
   }).$mount('#quickpay');   // can rename - must match div id below
}
```

This Angular component needs to include the implements OnInit on the class definition.

```
export class [AngularComponentName] implements OnInit {
…
```

1. Add to your Angular component's html markup.

```
<div id="quickpay">
  [any markup you want including any normal QuickPayments markup]
</div>
```

1. Angular will give you an error when in development since it won't recognize the QuickPaymentsJS markup. If you want to fix that, you can optionally add the **NO_ERRORS_SCHEMA** to the Angular Component's module.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule, NO_ERRORS_SCHEMA } from '@angular/core';
import { <AngularComponentName> } from './app.component';

@NgModule({
   declarations: [
      <AngularComponentName>
   ],
   imports: [
      BrowserModule
   ],
   providers: [],
   bootstrap: [<AngularComponentName>],
   schemas: [ NO_ERRORS_SCHEMA ]
})
export class <ComponentModule> { }
```

1. CSS. If you are already using FontAwesome (or other SVG font) and Bootstrap (or other CSS framework) in your app, you're good to go on that otherwise you will need to bring those in at an app or component level. In addition, you'll need the QuickPayments CSS. Bring that in at an app or component level also. At the app level you can just add it to the **<head>** in the index.html file.

```
<link rel="stylesheet" href="<path to>/quickpayments.css">
```

React

1. Download the minified QuickPaymentsJS library and minified VueJS library.
2. React has build issues with loading components that aren't in the src folder and other external components from React. This limit is only in the build process not in the functionality of either React or with QuickPaymentsJS. So, you need to load the files in the index.html and not at the component level.

```
<script src="%PUBLIC_URL%/vue.js"></script>
<script src="%PUBLIC_URL%/quickpayments.umd.min.js"></script>
```

1. Now, you can go to the file for the component in which you wish to use QuickPaymentsJS. At the top of that file, after all your imports, you need to let the linter know about a couple of variables. Do that by defining a couple of constants.

```
const Vue = window.Vue;
const quickpayments = window.quickpayments;
```

Now you can use these in your React component, inside the **componentDidMount()** lifecycle hook.

```
componentDidMount() {
   new Vue({
      components: {
         payment: quickpayments.payment
      }
   }).$mount('#quickpay-app'); // can rename - must match div id below
}
```

1. Add to your React component's JSX markup.

```
<div id="quickpay-app">
   [any markup you want including any normal QuickPayments markup]
</div>
```

1. Like Angular, React will give you an error (which really should only be a warning) in the browser's console when in development mode since it won't like the QuickPayments markup. Unlike Angular though, it appears there is no way to prevent that display. But, it only appears in development mode.
2. CSS. If you are already using FontAwesome (or other SVG font) and Bootstrap (or other CSS framework) in your app, you're good to go on that otherwise you will need to bring those in at an app or component level. In addition, you'll need the QuickPayments CSS. Bring that in at an app or component level also. At the app level you can just add it to the **<head>** in the index.html file.

```
<link rel="stylesheet" href="%PUBLIC_URL%/<path to>/quickpayments.css">
```