Hosted Payment Fields

Integration Guide – V1.0

Table of Contents

1 Hosted Payment Fields

1.1 About This Guide

This guide provides the information required to integrate with our Payment Gateway and gives a very basic example of code for doing so. It is expected that you have some experience in server-side scripting with languages such as PHP or ASP; or that an off-the-shelf software package is being used that has inbuilt or plug-in support for our Gateway.

Disclaimer

The Gateway provides all integration documentation necessary for enabling Merchants to process payments via our Payment Gateway. Whilst every effort has been made to ensure these guides are accurate and complete, we expect Merchants undertaking any integration to test all their technical work fully and satisfy their own standards. The Gateway is not responsible or liable for any Merchant or Third-Party integration.

1.2 Terminology

The following terms are used throughout this guide:

Gateway
The Payment Gateway.

Merchant
The Merchant using the Gateway’s services.

Our
The Payment Gateway Provider.

You/your
The Merchant or its representative performing the integration.

Acquirer
The bank or financial institution used by the Merchant.

Customer
A Customer of the Merchant making a payment.

Card
A payment credit, debit, prepayment or gift card issued by the Card Schemes.

Card Scheme
The operator of a payment Card network, such as Visa, Mastercard, et al.

Cardholder
The person who owns the payment Card, usually the Customer.

Issuer
The bank or financial institution that issued the payment Card to the Cardholder.

Merchant Account
An account on the Gateway mapped to an Acquirer-provided account.

Checkout
Third-party checkout solution such as PayPal, AmazonPay and other alternative payment methods.

Wallet
Third-party wallet solution such as Masterpass.

Hosted Payment Page (HPP)
A page hosted on our secure server used to collect Customer details.

Hosted Payment Field (HPF)
An individual form field hosted on our secure server used to collect sensitive Cardholder data.

1.3 Background

Hosted Payment Fields are a set of prebuilt JavaScript UI components that can be used by your website's HTML payment form to collect sensitive payment details without those details touching your server. They provide you with the PCI benefits of using a Hosted Payment Page, while allowing you the ability to design and implement your own payment forms.

There are 6 predefined Hosted Payment Fields available as follows:

- **cardNumber** – collects the card number.
- **cardCVV** – collects the card cvv.
- **cardExpiryDate** – collects the card expiry month and year.
- **cardStartDate** – collects the card start/issue month and year.
- **cardIssueNumber** – collects the card issue number.
- **cardDetails** – collects the card number, expiry date and cvv in a single field.

The **cardNumber** field is designed to collect a card number, including an icon used to display the card type. The field will only accept digits and spaces and validate that any entered value is a correctly formatted card number and insert spaces at the correct positions for the card type as the number is typed.

The **cardCVV** field is designed to collect a card CVV. The field will only accept digits and will validate that any entered value is a correctly formatted CVV, taking into account the card type as determined by an associated **cardNumber** field.

The **cardExpiryDate** and **cardStartDate** fields are designed to collect a card expiry date and card issue date respectively. The fields can render as a pair of select controls containing the months and a suitable range of years; or as an input control that will only allow digits to be entered and automatically formatted as a month / year entry. The field will validate that any entered value is a valid month and year combination.

The **cardIssueNumber** field is designed to collect a card issue number. The field will only accept digits and will validate that any entered value is a correctly formatted issue number.

The **cardDetails** field is designed to collect all of the essential card details. It combines the **cardNumber**, **cardExpiryDate** and **cardCVV** fields into a single line compound field design to allow easy entry of the card details and to complement the look of your checkout.

The field type is either: passed as the value of the **type** option the **Field** construction, provided by the HTML element's meta data; or provided via the HTML element's type attribute (prefixed with the 'hostedfield:' name space).

The following example shows all three approaches to specifying the field type:

```html
<input type="hostedfield:cardNumber" name="card-number">
<div class="hostedfield" data-hostedfield-type="cardExpiryDate"></div>
<input data-hostedfield='{"type":"cardCVV"}'>
```

It is highly recommended that you adopt a single approach as above and don't mix and match.

Each field type has its own additional configuration options, as detailed in section 2.5**.**

1.4 Pre-requisites

Hosted Fields requires knowledge of server-side scripting and HTML/CSS/JavaScript. Before you can integrate with Hosted Fields, you’ll need the following information:

- Merchant ID
- Hosted Integration URL or Direct Integration URL

2 Integration Details

2.1 Library Namespace

To avoid polluting the global namespace, the library extends the global window object with a **hostedFields** object containing the following properties:

- **forms** – array containing all the instantiated **Form** objects.
- **classes** – array containing all the instantiable classes.
    - **form** – **Form** class prototype.

2.2 Form Construction

The construction method can be used to prepare a HTML FORM for use with Hosted Payment Field components. The method signature is as follows:

Form(element, options)

The **element** parameter should be the DOM node of an existing FORM tag.

The **options** parameter should be object containing one of more of the following optional properties:

- **autoSetup** – boolean indicating whether setup should be handled automatically.
- **autoSubmit** – boolean indicating whether submission should be handled automatically.
- **merchantID** – string containing the **merchantID** the payment request is for.
- **stylesheet** – string containing DOM selector for any stylesheets to be used.
- **tokenise** – string/array/object specifying fields whose values should be tokenised.
- **fields** – object containing field configuration by field type.
- **locale** – string containing the desired locale.
- **classes** – object containing names of extra CSS classes to use.
- **submitOnEnter** – boolean indicating whether the enter key should cause the form to submit.
- **nativeEvents** – boolean indicating that native browser events should be fired.

Any **options** parameter will be merged with those provided via meta data supplied, using data-hostedfield and/or data-hostedfield-<*option*> attributes; or via existing attributes or properties of the **element**.

The **autoSetup** option can be used to disable the automatic creation of **Field** objects for the FORM child controls by calling the **autoSetup()** method during the **Form** construction. If automatic setup is disabled, then you must manually instantiate **Field** objects and attach them to the **Form** as required, using the **addField()** method. This option or manually calling the **autoSetup()** method minimises the amount of JavaScript you have to write. Automatic operation is good if you don't need to customise the operation or can't customise it by reacting to the **Form** or **Field** events. The option defaults to true and cannot be changed once the **Form** has been created.

The **autoSubmit** option can be used to disable the automatic handling of the FORM submission via the **autoSubmit()** method. If automatic submission is disabled, then you must manually retrieve the sensitive payment details by calling **getPaymentDetails()** and include them in the form submission data. This option or manually calling the **autoSubmit()** method minimises the amount of JavaScript you have to write. Automatic operation is good if you don't need to customise the operation or can't customise it by reacting to the **Form** or **Field** events. The option defaults to true and cannot be changed once the **Form** has been created.

The **merchantID** option can be used to specify the **merchantID** with which the final **paymentToken** will be used. The option defaults to the value of any child INPUT node whose name is ‘merchantID’ and can be changed at runtime by calling the **setMerchantID()** method or by altering the options using the jQuery **hostedForm()** plugin method.

The **stylesheet** option can be used to specify a DOM selector used to locate stylesheets that should be parsed for styles related to the Hosted Payment Fields. Refer to section 2.9 for how to style the Hosted Payment Fields using CSS stylesheets. The option defaults to the DOM selector string ‘link.hostedfield[rel=stylesheet], style.hostedfield' and can be changed at runtime by calling the **setStylesheet()** method; or by altering the options using the jQuery **hostedForm()** plugin method.

The **tokenise** option can be used to specify addition FORM controls whose values, as returned by the jQuery.val() method, should be included in the final **paymentToken**.
The option’s value must be either:

- A string containing a DOM selector used to select one or more controls.
- An array containing values used to jQuery.filter() down to one or more controls.
- An object whose properties are the name of fields to tokenise and whose values are objects containing a **selector** property used to select a control.

For the first two, the tokenised field’s name be will be taken from the controls data-hostedfield-tokenise attribute or name attribute. For the third, the name is property name in the **tokenise** object. If the field’s name is of the format ‘paymentToken[<name>]’, then only the ‘<name>’ part is used. The option defaults to the DOM selector string 'INPUT.hostedfield-tokenise:not(:disabled), INPUT[data-hostedfield-tokenise]:not(:disabled), INPUT[name^="paymentToken["]:not(:disabled)' and cannot be changed once the **Form** has been created.

The **fields** options can be used to specify default options for the different types of Hosted Payment Fields. The option’s value should be an object whose properties are the fields type or the wildcard type ‘any’ and whose values are objects whose properties are the default options for fields of that type. The values can also contain a **selector** property containing a DOM selector that is used during the automatic setup stage to select a FORM’s child element to add as a **Field** of the specified type automatically. The option has no default value and cannot be changed once the **Form** has been created.

The **locale** option can be used to specify the language that should be used by the Hosted Payment Fields attached to this **Form**. The option defaults to the value provided by any lang attribute on the **element** or closest ancestor and cannot be changed once the **Form** has been created.

The **classes** options can be used to specify additional CSS class names to add in addition to the default classes documented in section 2.8. The value is an object whose properties are the default class name and whose values are a string containing the additional class name(s) to use. The option has no default and cannot be changed once the **Form** has been created.

The **submitOnEnter** option can be used to specify if pressing the enter key when typing a **Field** value should cause the **Form** to submit. The option defaults to false and cannot be changed once the **Form** has been created.

The **nativeEvents** option can be used to specify that any associated native event should be fired when a ‘hostedField:’ prefixed **Field** event is fired (as documented in section 2.7). For example, when enabled if the ‘hostedfield:mouseover’ event is fired, then the native ‘mouseover’ event is also fired. The option defaults to false and cannot be changed once the **Form** has been created.

If not explicitly constructed, a **Form** object will be automatically instantiated and attached to the FORM DOM node as soon as any **Field** object is instantiated on a child DOM node.

2.3 Form Methods

The follow methods are made available by the **Form** class:

void autoSetup()

Automatically setup the form by scanning the Form element for child nodes to control as Hosted Payment Fields. Child nodes are selected if they:

- have a type attribute with a hostedfield:<*type*> value *(INPUT nodes only)*.
- have a data attribute with a hostedfield.<*type*> property.
- match a DOM selector provided by the fields*.*<*type*>.selector option.

If multiple selection criteria are present, then they must all specify the same **Field** type or an exception is thrown.

This method is called during the **Form** construction unless the **autoSetup** option is false.

void autoSubmit()

Automatically handles any attempted FORM submission by checking the FORM's controls are valid by calling the **validate()** method; and then requesting the **paymentToken** using the **getPaymentDetails()** method; and finally adding the token to the forms fields using the **addPaymentToken()** method. Failure to validate or request the payment token will cause the form submission to be stopped.

You can affect the automatic submission stages by listening for events and preventing their default actions. The full list of events is documented in section 2.7.

This method is attached to the FORM submit event during the **Form** construction unless the **autoSubmit** option is false, or the **autoSubmit** option is null and the **autoSetup** option is false.

If automatic submission is disabled, then you must react to the FORM's submit event and then request the **paymentToken** using the **getPaymentDetails()** method and ensure that the token is sent as part of the form's data.

boolean addField(Field f)

Add a hosted **Field** to the Form.

Returns true if successful, false otherwise.

boolean delField(Field f)

Remove a hosted **Field** from the **Form**.

Returns true if successful, false otherwise.

promise validate(boolean submitting)

Validate all **Field** values on the **Form**, either during submission or not.

Returns a promise that will be resolved when the validation is complete.

object[] getInvalidElements()

Get details about all invalid FORM controls (not just invalid hosted **Field** elements).

Returns an array of objects containing the following properties:

- **element** – DOM element.
- **message** – DOM elements validationMessage property or ‘Invalid value’.
- **label** – associated LABEL text.
- **field** – **Field** instance (if DOM element is a hosted **Field**).

object getValidationErrors()

Get the validation errors for all invalid FORM controls (not just invalid hosted **Field** elements).

Returns an object whose properties are the associated labels, names or id of the invalid FORM controls and whose values are the error message for that control.

promise getPaymentDetails(object tokenData, boolean validate)

Gets the payment details, generating a **paymentToken** containing the hosted Field values; any values specified by the **tokenise** option; and any passed **tokenData**. The Form will be validated first if required.

Returns a promise that will be resolved when the payment details have been obtained, passing the details as an object containing the following properties:

- **success** – boolean true if successful, false otherwise.
- **message** – string containing message to display if not successful.
- **errors** – object containing details about invalid payment data.
- **invalid** – object as returned by **getValidationErrors()** method.
- **paymentToken** – string containing generated **paymentToken**.

void addPaymentToken(string token)

Add the payment token as the value of a Form child INPUT whose name is ‘paymentToken’, creating the control if needed. Any created control will be given a type of ‘hidden’.

void setMerchantID(string merchantID)

Set the **merchantID** used by the payment form.

void setStylesheet(string selector)

Set the DOM selector used to select the stylesheet(s) used by the **Form**.

object defaultFieldOptions(string type)

Get any default field options specified via the **fields** option, resulting from the merger of its optional **any** and ***<type>*** properties.

Returns an object whose properties are the default options.

void forceSubmit()

Forcefully submit the FORM **element** as if a child submit button had been clicked.

void reset()

Reset all the **Form**, setting all **Field** values back to their initial values.

void destroy()

Destroys the **Form,** reverting its **element** back to its original state.

2.4 Form Events

The following events may be fired by the **Form** object and you can use these to hook into and modify the object's behaviour:

Event Name[1]	**Description**
`create`	Fired when a **`Form`** has been created.
`destroy`	Fired when a **`Form`** has been destroyed.
`presubmit`	Fired by the **`autoSubmit()`** method prior to handling the submission. You can prevent the handling of the submission and handle it yourself by calling the Events `preventDefault()` method.
`valid`	Fired by the **`autoSubmit()`** method if the FORM contains valid data prior to requesting the payment details. You can prevent the continued handling of the submission and handle it yourself by calling the Events `preventDefault()` method or by invalidating the FORM.
`submit-invalid`	Fired by the **`autoSubmit()`** method if the FORM contains invalid data prior to displaying the validity using the DOM `reportValidity()` method. You can prevent the `reportValidity()` call and display the validity yourself by calling the Events `preventDefault()` method.
`submit`	Fired by the **`autoSubmit()`** method prior to submitting the FORM. You can prevent the FORM from submitting by calling the Events `preventDefault()` method.
`error`	Fired by the **`autoSubmit()`** method if an exception is caught prior to displaying the error, using the JavaScript `alert()` function. You can prevent the `alert()` call and display the error yourself by calling the Events `preventDefault()` method.

[1] Event names are prefixed with the 'hostedform:' namespace not shown in the table.

The **presubmit**, **valid**, **submit-invalid**, **submit** and **error** events fired by the **autoSubmit()** method payload is an object with the following properties:

- **success** – boolean false.
- **message** – error message if **error** otherwise null.
- **invalid** – result of **getValidationErrors()** method if **Form** invalid.
- **submitting** – boolean true.

2.5 Field Construction

The construction method can be used to prepare a HTML INPUT control as a Hosted Payment Field or to create a new field in HTML DIV container. The method signature is as follows:

```
Field(element, options)
```

The **`element`** parameter should be the DOM node of an existing INPUT or DIV tag.

The **`options`** parameter should be an object containing one of more of the following optional properties:

- **`type`** – string containing the desired field type.
- **`value`** – string containing the initial value.
- **`placeholder`** – string containing any placeholder text.
- **`style`** – string containing any inline CSS styles.
- **`stylesheet`** – string containing DOM selector for any stylesheets to be used.
- **`disabled`** – boolean indicating if initially disabled.
- **`required`** – boolean indicating if the value is required.
- **`readOnly`** – boolean indicating if initially read only.
- **`validity`** – boolean or string indicating the initial validity.
- **`locale`** – string containing the desired locale.
- **`classes`** – object containing names of extra CSS classes to use.
- **`submitOnEnter`** – boolean indicating if the enter key should cause the form to submit.
- **`nativeEvents`** – boolean indicating that native browser events should be fired.
- **`validationMessages`** – object containing alternative validation messages.
  - **`required`** – string containing validation message to use when a value is required.
  - **`invalid`** – string containing validation message to use when a value is invalid.
- **`format`** – string containing select option format for date fields.
- **`minYear`** – integer containing minimum year (relative to current year) for date fields.
- **`maxYear`** – integer containing maximum year (relative to current year) for date fields.

Any **`options`** parameter will be merged with those provided via meta data supplied using `data-hostedfield` and/or `data-hostedfield-`*`<option>`* attributes, or via existing attributes or properties of the **`element`** or provided via the **`getDefaultOptions()`** method of the parent **`Form`**.

The **`type`** option can be used to specify the type of Hosted Payment Field required. It defaults to the value provided by any `type` attribute on the **element** (prefixed with the 'hostedfield:' namespace). The option cannot be changed once the **`Field`** has been created. Valid types are **cardDetails**, **cardNumber**, **cardCVV**, **cardExpiryDate**, **cardStartDate**, **cardIssueNumber**.

The **`value`** option can be used to specify any initial value that should be used by the **`Field`**. It defaults to the value provided by any `value` attribute or property on the **element**. Obviously, due to the purpose of the Hosted Payment Fields, any initial value is not wise for card number and CVV fields. The option can be changed at runtime by calling the **`setValue()`** method.

The **placeholder** option can be used to specify any initial text that should be used as a placeholder by the **Field**. It defaults to the value provided by any placeholder attribute or property on the **element**. When used with the **CardDetails** type **Field** the placeholder contains three parts separated by a pipe character, the first part contains the **cardNumber** placeholder, the second part contains the **cardExpiry** placeholder, and the third part contains the **cardCVV** placeholder. The option can be changed at runtime by calling the **setPlaceholder()** method or by altering the options using the jQuery **hostedForm()** plugin method.

The **style** option can be used to specify any initial inline CSS style that should be used by the **Field**. It defaults to the value provided by any style attribute or property on the **element**. The option can be changed at runtime by calling the **setStyle()** method or by altering the options using the jQuery **hostedForm()** plugin method.

The **stylesheet** option can be used to specify a DOM selector used to locate stylesheets that should be parsed for styles related to this **Field**. Refer to section on styling fields. The option can be changed at runtime by calling the **setStylesheet()** method or by altering the options using the jQuery **hostedForm()** plugin method.

The **disabled** option can be used to specify if the **Field** should be initially disabled. It defaults to the value provided by any disabled attribute or property on the **element**. The option can be changed at runtime by calling the **setDisabled()** method or by altering the options using the jQuery **hostedForm()** plugin method.

The **required** option can be used to specify if the **Field** value is required. It defaults to the value provided by any required attribute or property on the **element**. The option can be changed at runtime by calling the **setRequired()** method or by altering the options using the jQuery **hostedForm()** plugin method.

The **readOnly** option can be used to specify if the **Field** should be initially read-only. It defaults to the value provided by any readOnly attribute or property on the **element**. The option can be changed at runtime by calling the **setReadOnly()** method or by altering the options using the jQuery **hostedForm()** plugin method.

The **validity** option can be used to specify if the **Field** should be initially marked as invalid. It defaults to the value provided by any validity property on the **element**. The option can be changed at runtime by calling the **setValidity()** method or by altering the options using the jQuery **hostedForm()** plugin method.

The **locale** option can be used to specify the language that should be used by the **Field**. It defaults to the value provided by any lang attribute or property on the **element** or closest ancestor. The option cannot be changed once the **Field** has been created.

The **classes** options can be used to specify additional CSS class names to add in addition to the default classes documented in section 2.8. The value is an object whose properties are the default class name and whose values are a string containing the additional class name(s) to use. This option will be merged with any classes option provided to the **Form** constructor. The option cannot be changed once the **Form** has been created.

The **submitOnEnter** option can be used to specify if pressing the enter key when typing the **Field** value should cause the **Form** to submit. The option defaults to false and cannot be changed once the **Field** has been created.

The **nativeEvents** option can be used to specify that any associated native event should be fired when a ‘hostedfield:’ prefixed event is fired. Events are documented in section 2.7. For example, when enabled, if the ‘hostedfield:mouseover’ event is fired then the native ‘mouseover’ event is also fired. The option defaults to false and cannot be changed once the **Field** has been created.

The **validationMessages** option can be used to specify alternative validation messages that should be displayed when a value is required or invalid. The option defaults to suitable messages depending on the locale and cannot be changed once the **Field** has been created.

The **dropdown** option can be used to specify that a **cardStartDate** or **cardExpiryDate Field** should be displayed as a pair of select controls to select the month and year, otherwise the month and year are entered via a formatted input box instead. The option defaults to false and cannot be changed once the **Field** has been created.

The **format** option can be used in conjunction with the **dropdown** option to specify the format used to display the month and year in the dropdowns. The month and year parts of the format are separated by a pipe character. The option defaults to ‘N – M | Y’ (e.g. ‘01 – January | 2020’) and cannot be changed once the **Field** has been created.

The following formatting characters are understood:
- **n** – month number (no zero prefixing).
- **N** – month number (zero prefixed to two digits when required).
- **m** – short month name (e.g. Jan, Feb, Mar)
- **M** – long month name (e.g. January, February, March)
- **y** – two-digit year number.
- **Y** – four-digit year number.

The **minYear** and **maxYear** options can be used in conjunction with the **dropdown** option to specify the minimum and maximum years that are included in the year dropdown. The option defaults to minus 20 to zero for a **cardStartDate Field** or zero to plus 20 for a **cardExpiryDate Field** and cannot be changed once the **Field** has been created.

2.6 Field Methods

The follow methods are made available by the `Field` class:

`promise validate()`

Validate the `Field` value. This will normally be called automatically when the `Field` loses focus or the form is submitted, or when an invalid value is modified.

Returns a `promise` that will be resolved when the validation is complete.

`boolean isEmpty()`

Check if the `Field` has a value.

Returns true if the field has a value, false otherwise.

`boolean isComplete()`

Check if the `Field` has a complete, but not necessarily valid, value. This is mainly used by compound fields such as **cardDetails**, **cardExpiryDate**, **cardStartDate**, which contain multiple input controls and are deemed complete when all their required input controls have values.

Returns true if the value is complete, false otherwise.

`void setStyle()` / **`string getStyle()`**

Set or gets the field's inline CSS style data.

Returns void when setting, or a CSS style string when getting.

`void setStylesheet(string selector)` / **`string getStylesheet()`**

Sets or gets the DOM selector used to select the stylesheet(s) used by the `Field`. When setting, the stylesheets are parsed and applied to the `Field`.

Returns void when setting, or a DOM selector string when getting.

`void setPlaceholder(string text)` / **`string getPlaceholder()`**

Sets or gets the placeholder text to be shown when the `Field` has no value.

When used with the **CardDetails** type `Field` the placeholder contains three parts separated by a pipe character, the first part contains the **cardNumber** placeholder, the second part contains the **cardExpiry** placeholder, and the third part contains the **cardCVV** placeholder.

Returns void when setting, or a text string when getting.

`void setDisabled(boolean disabled)` / `string getDisabled()`

Sets or gets the disabled state of the **`Field`**. When disabled, the field will be greyed out and not be focusable and thus will not react to any input events.

A disabled **`Field`** will have the ‘hf-disabled’ class added otherwise the ‘hf-enabled’ class is added.

Returns void when setting, or a boolean representing the state when getting.

`void setRequired(boolean required)` / `string getRequired()`

Sets or gets the required state of the **`Field`**. When required, the field will be invalid if it contains no value or a blank value.

A required **`Field`** will have the ‘hf-required’ class added otherwise the ‘hf-optional’ class is added.

Returns void when setting, or a boolean representing the state when getting.

`void setReadOnly(boolean read_only)` / `string getRequired()`

Sets or gets the read-only state of the **`Field`**. When read-only, the field will be not be focusable and thus will not react to any input events.

A read-only **`Field`** will have the ‘hf-readonly’ class added otherwise the ‘hf-readwrite’ class is added.

Returns void when setting, or a boolean representing the state when getting.

`void setFocused(boolean focused)`

Moves the browser’s focus to the **`Field`**. When focused, the field will react input events.

A focused **`Field`** will have the ‘hf-focus’ class added otherwise the ‘hf-blur’ class is added.

Returns void when setting, or a boolean representing the state when getting.

`void setValidity(string validity)` / `string getValidity()`

Sets or gets the validity of the **`Field`**. When valid, the validity will be true or a blank string. When invalid, the validity will be an error message explaining the reason the value is invalid.

When used with the **CardDetails** type **`Field`** the error message contains three parts separated by a pipe character, the first part contains the **cardNumber** value, the second part contains the **cardExpiry** value, and the third part contains the **cardCVV** value.

A valid **`Field`** will have the ‘hf-valid’ and ‘hf-user-valid’ classes added otherwise the ‘hf-invalid’ and ‘hf-user-invalid’ classes are added.

Returns void when setting, or an error message string when getting.

`void setValue()` / **`string getValue()`**

Set or gets the **`Field`** value. Because Hosted Payment Fields are designed for the entry of sensitive payment details, then these methods are not normally used. There is no means to retrieve the actual sensitive data and so any returned value will be an empty string if the field has no value or a single asterisk if the field has a value.

When used with the **CardDetails** type **`Field,`** the value contains three parts separated by a pipe character, the first part contains the **cardNumber** value, the second part contains the **cardExpiry** value, and the third part contains the **cardCVV** value.

Returns void when setting, or a mask string when getting.

`void getState()`

Get the current state of the **`Field`** as an object with the following boolean properties:

- **`isReady`** – the **`Field`** has been created, initialised and is ready for use.
- **`isValid`** – the value is valid (refer to the **`setValidity()`** method).
- **`isEmpty`** – the value is empty (refer to the **`isEmpty()`** method).
- **`isComplete`** – the value is complete (refer to the **`isComplete()`** method).
- **`isDisabled`** – the value is complete (refer to the **`setDisabled()`** method).
- **`isRequired`** – the value is complete (refer to the **`setRequired()`** method).
- **`isReadOnly`** – the value is complete (refer to the **`setReadOnly()`** method).

Returns an object containing the states.

`void reset()`

Reset **`Field`** value back to the initial value.

`void destroy()`

Destroys the **`Form,`** reverting its **`element`** back to its original state.

Note: A field’s options or properties cannot be changed while a field is initialising: that is between construction and firing of the ‘ready’ event. Attempts to change field options or properties before this will be ignored.

2.7 Field Events

The following events may be fired by the **Field** object and you can use these to hook into and modify the object's behaviour:

Event Name[1]	Description
`create`	Fired when a `Field` has been created.
`destroy`	Fired when a `Field` has been destroyed.
`ready`	Fired when a `Field` style is has finished initialising and is ready.
`style`	Fired when a `Field` style is changed.
`autofill`	Fired when a `Field` has a value auto filled by the browser.
`autofillcancel`	Fired when a `Field` has an auto filled value removed.
`valid`	Fired when a `Field` is checked for validity and passes the check.
`invalid`	Fired when a `Field` is checked for validity and fails the check.
`uservalid`	Fired when the valid event is fired but only after user interaction has occurred, such as focusing a `Field`, leaving a `Field` or attempting to submit a `Form`.
`userinvalid`	Fired when the invalid event is fired but only after user interaction has occurred, such as focusing a `Field`, leaving a `Field` or attempting to submit a `Form`.
`disabled`	Fired when a `Field` changes to disabled.
`enabled`	Fired when a `Field` changes from disabled.
`required`	Fired when a `Field` changes to required.
`optional`	Fired when a `Field` changes from required.
`readonly`	Fired when a `Field` changes to read-only.
`readwrite`	Fired when a `Field` changed from read-only.
`focus`	Fired when a `Field` receives focus.
`blur`	Fired when a `Field` loses focus.
`mouseenter`	Fired when a pointing device is moved into the `Field`.
`mouseleave`	Fired when a pointing device is moved out of the `Field`.
`mouseover`	Fired when a pointing device is moved into the `Field`.
`mouseout`	Fired when a pointing device is moved out of the `Field`.
`mousemove`	Fired when a pointing device is moved over the `Field`.
`keydown`	Fired when a key is pressed in the `Field`.
`keyup`	Fired when a key is released in a `Field`.
`keypress`	Fired when a key except Shift, Fn, CapsLock is in a pressed position in a `Field`.

`change`	Fired when an alteration to the value of a **`Field`** is committed by the user.
`input`	Fired when the value of a **`Field`** is changed.

[1] Event names are prefixed with the ‘hostedfield:’ namespace not shown in the table.

2.8 Field CSS Classes

The following CSS class names will be added to a **`Field`** object depending on its state and you can use these to style the object as required:

Event Name	Description
`hostedfield`	Present on all **`Field elements`**.
`hf-autofill`	Present when the value was auto filled by the browser.
`hf-invalid`	Present when in the invalid state.
`hf-valid`	Present when in the valid state.
`hf-user-invalid`	Present when in the invalid state and user interaction has occurred, such as focusing a **`Field`**, leaving a **`Field`** or attempting to submit a **`Form`**.
`hf-user-valid`	Present when in the valid state and user interaction has occurred, such as focusing a **`Field`**, leaving a **`Field`** or attempting to submit a **`Form`**.
`hf-disabled`	Present when in the disabled state.
`hf-enabled`	Present when not in the disabled state.
`hf-required`	Present when in the required state.
`hf-optional`	Present when not in the required state.
`hf-readonly`	Present when in the read-only state.
`hf-readwrite`	Present when not in the read-only state.
`hf-focus`	Present when in the focused state.
`hf-blur`	Present when not in the focused state.
`hf-empty`	Present when in the empty state.
`hf-complete`	Present when in the complete state.
`hf-hover`	Present when a pointing device is over the **`Field`**.
`hf-placeholder-shown`	Present when the placeholder text is displayed.

In addition to these class names, the **`Field`** will add any corresponding class names provided by the **`classes`** option provided when the **`Field`** is constructed.

For example if the **`Field`** is constructed with a **`classes`** option as follows: '`{disabled: 'text-blur text-grey', enabled: 'text-normal'}`', then the 'text-blur' and 'text-grey' class names will be present whenever the 'hf-disabled' class is present and the 'text-normal' class name will be present whenever the 'hf-enabled' class name is present.

2.9 Field Styling

The Hosted Payment Fields are styled using CSS as normal.

However, styles must be transferred from your website to the controls served from our website, therefore styles must be isolated and easily identifiable. To aid with identification, all styles intended for a **`Field`** must contain the ‘hostedfield’ class name in their selector or ‘-hostedfield’ extension on any id in the selector.

As a website may contain lots of stylesheets, a **`Field`** cannot be expected to parse every stylesheet present on the page and therefore it only parses those selected using the stylesheets construction option or using the **`setStylesheet()`** method. By default, this is any stylesheet referenced via a <link> tag or <style> tag with the ‘hostedfield’ class name: i.e. any HTML node that matches the following DOM selector ‘link.hostedfield[rel=stylesheet], style.hostedfield'.

CSS styles using the **`Field`** state classes, pseudo classes and pseudo elements are supported as follows:

- focus, .hf-focus,
- hover, .hf-hover,
- enabled, .hf-enabled,
- disabled, .hf-disabled,
- valid, .hf-valid,
- invalid, .hf-invalid,
- user-valid, .hf-user-valid,
- user-invalid, .hf-user-invalid,
- required, .hf-required
- optional, .hf-optional,
- empty, .hf-empty,
- complete, .hf-complete,
- autofill, .hf-autofill,
- placeholder-shown, .hf-placeholder-shown,
- readonly, .hf-readonly,
- readwrite, .hf-readwrite,
- -webkit-auto-fill, .hf-icon,
- :placeholder,
- :-moz-placeholder,
- :-webkit-input-placeholder,
- :-ms-input-placeholder

The styles can contain any valid CSS rules and will be used to style both the public elements and internal private elements. For security only, styles that relate to the textual representation of the **`Field`** are passed to the internal private elements. This include styles such as colours, font weights and text decorations. At present, it is not possible to specify custom fonts as they would require the font files to be available on our servers.

The following styles can be used to style the **`Field`** internal private elements:

> caret-color, color, cursor, direction, fill, filter, font, font-family, font-feature-settings, font-kerning, font-language-override, font-size, font-size-adjust, font-smooth, font-stretch, font-style, font-synthesis, font-variant, font-variant-alternates, font-variant-caps, font-variant-east-asian, font-variant-ligatures, font-variant-numeric, font-variant-position, font-weight, letter-spacing, line-height, stroke, text-align, text-decoration, text-decoration-color, text-decoration-line, text-decoration-style, text-emphasis, text-emphasis-color, text-emphasis-position, text-emphasis-style, text-indent, text-rendering, text-shadow, text-transform, text-underline-position, -moz-osx-font-smoothing, -webkit-font-smoothing, -webkit-text-fill-color

The ‘.hf-icon’ class name can be used to target the icon sub element in a **`cardDetails`** **`Field`**.

Individual controls can be targeted by using DOM ids, which will have a ‘-hostedfield’ extension added to the DOM id of the original **`element`**.

It is advisable to keep CSS selectors and rules as simple as possible to avoid styling errors caused by a failure to parse and filter the rules.

2.10 jQuery Plugin

The script will extend the jQuery object with its own plugin methods to allow easy access to **`Form`** and **`Field`** objects attached to an **`element`** as follows:

```
$(element).hostedForm(options);
$(element).hostedForm(‘instance’);
$(element).hostedForm(‘options’, options);
$(element).hostedForm(method, parameters);
$(element).hostedForm(‘destroy’);

$(element).hostedField(options);
$(element).hostedField(‘instance’);
$(element).hostedField(‘options’, options);
$(element).hostedField(method, parameters);
$(element).hostedField(‘destroy’);
```

The script will also add a ‘:hostedfield’ pseudo selector allowing **`Field`** elements to be selected using the following example notation:

```
$(‘INPUT:hostedfield’)
```

A1 Example Form Construction

The following shows an example **`Form`** construction:

```
var form = new window.hostedFields.classes.Form(document.forms[0],{
    // Auto setup the form creating all hosted fields (default)
    autoSetup: true,

    // Auto validate, tokenise and submit the form (default)
    autoSubmit: true,

    // Additional fields to tokenise
    tokenise: '.add-to-token',

    // Stylesheet selection
    stylesheets: '#hostedfield-stylesheet',

    // Optional field configuration (by type)
    fields: {
        any: {
            nativeEvents: true
        },
        cardNumber: {
            selector: $('#form2-card-number'),
            stylesheet: $('style.hostedform, style.hostedform-card-number')
        }
    },

    // Additional CSS classes
    classes: {
        invalid: 'error'
    }
});
```

Or using meta data on the HTML FORM element:

```
<form data-hostedfields='{"autoSetup":true,"autoSubmit":true,"tokenise":".add-to-
token","stylesheets":"#hostedfield-
stylesheet","fields":{"any":{"nativeEvents":true},"cardNumber":{"selector":"#form2-card-
number","stylesheet":"style.hostedform, style.hostedform-card-
number"}},"classes":{"invalid":"error"}}' method="post" novalidate="novalidate" lang="en">
<script>
var form = new window.hostedFields.classes.Form{document.forms[0]);
</script>
```

A2 Example Field Construction

The following shows an example **`Field`** construction:

```
var field = new window.hostedFields.classes.Field(document.forms[0].elements[0], {
    // Field type
    type: 'cardNumber',

    // Stylesheet selection
    stylesheets: '#hostedfield-stylesheet',

    // Additional CSS classes
    classes: {
        invalid: 'error'
    }
});
```

Or using meta data on the HTML INPUT element:

```
<input type="hostedfield:cardNumber" data-hostedfields='{"stylesheet":"style.hostedform, style.hostedform-card-
number"}},"classes":{"invalid":"error"}}'>
<script>
var field = new window.hostedFields.classes.Field{document.forms[0].elements[0]);
</script>
```

A3 Example Stylesheet

The following shows an example **`Stylesheet`** construction:
:

```
<style class="hostedfield">
    /*
     * Style hosted field internals
     * - only accept font, foreground and background styling
     */

    /* Copy of Bootstrap styles */
    .hostedfield:disabled {
        cursor: not-allowed;
        background-color: #eee;
        opacity: 1;
    }

    /* Change text to red when invalid */
    .form-control:invalid,
    .hostedfield:invalid {
        border-color: #a94442 !important;
        color: #a94442 !important;
    }

    /* Change text to light grey when readonly */
    .form-control:readonly,
    .hostedfield:readonly {
        color: lightgrey !important;
    }

    /* Emulate webkit auto fill style */
    .form-control.hf-autofill,
    .hostedfield.hf-autofill {
        background-color: rgb(250, 255, 189) !important;
        background-image: none !important;
        color: rgb(0, 0, 0) !important;
    }

    /* Add pink placeholder */
    .form-control::placeholder,
    .hostedfield::placeholder {
        color: pink;
    }

    /* Show hovering over the control */
    .form-control.hf-hover,
    .hostedfield.hf-hover {
        font-style: italic;
    }

    /* Style by id (hosted field will have '-hostedfield' appended to the id) */
    #form1-card-details.hostedfield, #form1-card-details-hostedfield {
        color: blue;
    }

</style>
```

A4 Hosted Payment Fields Library

The following example code shows how to create and manage Hosted Payment Fields using the Hosted Payment Field library.

The example shows how to style fields using an inline stylesheet and how to listen and react to the field's events.

The example also shows how to set up the payment form both automatically and manually and integrate with the jQuery validator plugin. You should choose the set-up method best suited for your needs and whatever validation plugin or functions you are familiar with.

*Note: The example code demonstrates including the static transaction information, such as the **`merchantID`** and **`amount`**, in hidden form fields and POSTing the form directly to the Gateway's Direct Integration using partial message signing. We would, however, recommend that you capture just the information you require and then POST this data to your own website where you can use it to build a new fully signed request to send to the Gateway's Direct Integration as a server to server request.*

```
<html>
  <head>
    <!-- Load the jQuery library -->
    <script src="https://code.jquery.com/jquery-3.4.1.min.js" integrity="sha256-
CSXorXvZcTkaix6Yvo6HppcZGetbYMGWSFlBw8HfCJo=" crossorigin="anonymous"></script>

    <!-- Load the jQuery Validator plugin -->
    <script src="https://cdn.jsdelivr.net/npm/jquery-validation@1.19.1/dist/jquery.validate.min.js"></script>

    <!-- Load the Hosted Payment Field library -->
    <script src="https://gateway.example.com/sdk/web/v1/js/hostedfields.js"></script>

    <!-- General styles -->
    <style>
      body {
        font-size: 14px;
      }

      .form-group {
        margin: 4px 0 15px 0;
      }

      .form-group LABEL {
        display: inline-block;
        max-width: 100%;
        margin-bottom: 5px;
        font-weight: bold;
      }

      .form-control {
        display: block;
        box-sizing: border-box;
        height: 34px;
        width: 400px;
        padding: 6px 12px;
        font-size: 14px;
        color: #555;
        background-color: #fff;
        background-image: none;
        border: 1px solid #ccc;
        border-radius: 4px;
        -webkit-box-shadow: inset 0 1px 1px rgba(0, 0, 0, .075);
        box-shadow: inset 0 1px 1px rgba(0, 0, 0, .075);
        -webkit-transition: border-color ease-in-out .15s, -webkit-box-shadow ease-in-out .15s;
        -o-transition: border-color ease-in-out .15s, box-shadow ease-in-out .15s;
        transition: border-color ease-in-out .15s, box-shadow ease-in-out .15s;
      }

      .form-control.hf-focus {
        border-color: #66afe9;
        outline: 0;
        -webkit-box-shadow: inset 0 1px 1px rgba(0,0,0,.075), 0 0 8px rgba(102,175,233,.6);
        box-shadow: inset 0 1px 1px rgba(0,0,0,.075), 0 0 8px rgba(102,175,233,.6);
      }

      .has-error .form-control.hf-focus {
        border-color: #843534;
        -webkit-box-shadow: inset 0 1px 1px rgba(0,0,0,.075), 0 0 6px #ce8483;
        box-shadow: inset 0 1px 1px rgba(0,0,0,.075), 0 0 6px #ce8483;
      }
    </style>

    <!-- Hosted Field internal styles -->
    <style class="hostedfield">
      /* Grey out when disabled */
      .hostedfield:disabled {
        cursor: not-allowed;
        background-color: #eee;
        opacity: 1;
      }

      /* Change border and text to green when valid */
      .form-control:valid,
      .hostedfield:valid {
        border-color: #28a745 !important;
        color: #28a745 !important;
      }

      /* Change border and text to red when invalid */
      .form-control:invalid,
      .hostedfield:invalid {
        border-color: #a94442 !important;
        color: #a94442 !important;
      }

      /* Change text to light grey when readonly */
      .form-control:readonly,
      .hostedfield:readonly {
        color: lightgrey !important;
      }

      /* Emulate webkit auto fill style */
      .form-control.hf-autofill,
      .hostedfield.hf-autofill {
        background-color: rgb(250, 255, 189) !important;
        background-image: none !important;
        color: rgb(0, 0, 0) !important;
      }

      /* Add light blue placeholder */
      .form-control::placeholder,
      .hostedfield::placeholder {
        color: lightblue;
      }

      /* Show hovering over the control */
      .form-control:hover,
      .hostedfield:hover {
        font-style: italic;
      }

      /* Style by id (hosted field will have '-hostedfield' appended to the id) */
      #form-card-number, #form-card-number-hostedfield {
        color: darkcyan;
      }

```

```html
    </style>

    <!-- Hosted Field card-number internal styles -->
    <style class="card-number">

      .hostedfield::placeholder {
        color: orange;
      }

    </style>
  </head>

  <body>
    <!-- tokenise payment data and send directly to the Gateway -->
    <form id="form" method="POST" novalidate="novalidate" lang="en"
      action="https://gateway.example.com/direct/"
      data-hostedform-tokenize='{"#form-customer-name": "customerName"}'>
      <input type="hidden" name="merchantID" value="100001">
      <input type="hidden" name="action" value="SALE">
      <input type="hidden" name="type" value="1">
      <input type="hidden" name="countryCode" value="826">
      <input type="hidden" name="currencyCode" value="826">
      <input type="hidden" name="amount" value="1001">
      <input type="hidden" name="orderRef" value="Test purchase">
      <input type="hidden" name="transactionUnique" value="1234">
      <input type="hidden" name="redirectURL" value="https://www.merchant.com/payment/">
      <input type="hidden" name="signature" value="5a0dd6fed71ef68bb3f20175b6a04bbd9d1c904d32ae3f160bd3b8f55740207e5d1e8de5e7e9960b136407e7454b82e428b8378003aa0146df3efa91a3e61b17|merchantID,action,type,countryCode,currencyCode,amount,orderRef,transactionUnique,redirectURL">
      <input type="hidden" name="paymentToken" value="">

      <div class="form-group">
        <label for="form-customer-name">Name on card:</label>
        <input id="form-customer-name" type="text" name="paymentToken[customerName]" autocomplete="cc-name" class="form-control form-control-native hostedfield-tokenise" placeholder="Firstname Surname" required>
      </div>

      <div class="form-group">
        <label for="form-card-number">Card Number:</label>
        <input id="form-card-number" type="hostedfield:cardNumber" name="card-number" autocomplete="cc-number" class="form-control form-control-hosted" style="background: #f2f8fb;" placeholder="**** **** **** ****" required>
      </div>

      <div class="form-group">
        <label for="form-card-expiry-date">Card Expiry Date:</label>
        <input id="form-card-expiry-date" type="hostedfield:cardExpiryDate" name="card-expiry-date" autocomplete="cc-exp" class="form-control form-control-hosted" required>
      </div>

      <div class="form-group">
        <label for="form-card-start-date">Card Issue Date:</label>
        <input id="form-card-start-date" type="hostedfield:cardStartDate" name="card-start-date" autocomplete="cc-iss" class="form-control form-control-hosted" data-hostedfield='{"dropdown":true}' data-hostedfield-format="N - m | y" data-hostedfield-min-date="-40" data-hostedfeld-max-date="0">
      </div>

      <div class="form-group">
        <label for="form-card-cvv">CVV:</label>
        <input id="form-card-cvv" type="hostedfield:cardCVV" name="card-cvv" autocomplete="cc-csc" class="form-control form-control-hosted" required>
      </div>

      <button id="form-submit" type="submit">Pay <span>▷</span></button>
    </form>

    <script>
      // This example demonstrates both automatic and manual form setup
      var automatic_setup = true;

      $(document).ready(function () {
```

```

      var $form = $('#form');

      // Listen for events on the form to see those sent from the Hosted Payment Fields
      // (For demonstration purposes only)
      $form.on(events);

      if (automatic_setup) {
        ///////////////////////////////////////////////////////////////
        // FORM AUTOMATIC SETUP
        ///////////////////////////////////////////////////////////////

        var opts = {
          // Auto setup the form creating all hosted fields (default)
          autoSetup: true,

          // Auto validate, tokenise and submit the form (default)
          autoSubmit: true,

          // Optional field configuration (by type)
          fields: {
            any: {
              nativeEvents: true,
            },
            cardNumber: {
              selector: $('#form-card-number'),
              style: 'text-decoration: green wavy underline;',
              stylesheet: $('style.hostedfields, style.card-number')
            }
          }
        };

        try {
          // Create form, automatically creating all child Hosted Payment Fields
          $form.hostedForm(opts);
        } catch(e) {
          showError('Failed to create hosted form: ' + e);
          throw e; // Can't continue with this script
        }

        // Listen for some events from the form thrown by the auto methods
        $form.on({
          // Let jQuery Validator check the form on submission
          'hostedform:presubmit': function (event) {
            console.log('Form submitting');
            return $form.valid();
          },

          // Show form is valid
          'hostedform:valid': function (event) {
            console.log('Form valid');
            return true;
          },

          // Show any validation errors
          'hostedform:invalid': function (event, details) {
            console.log('Form invalid');
            showFieldErrors(details.invalid);
            return true;
          },

          // Show general error
          'hostedform:error': function (event, details) {
            showError(details.message);
            return true;
          }
        });

        // Use jQuery validator to validate the form
        $form.validate();

        // End of form automatic setup

      } else {
        ///////////////////////////////////////////////////////////////
        // FORM MANUAL SETUP
        ///////////////////////////////////////////////////////////////

        try {
          // Create the card number field with custom options
          $('#form-card-number').hostedField({
            nativeEvents: true,
            style: 'text-decoration: green wavy underline;',
            stylesheet: $('style.hostedfields, style.card-number')
          });

          // Create the remaining hosted fields
          $('.form-control-hosted:input', $form).hostedField({nativeEvents: true});

        } catch (e) {
          showError('Failed to create hosted fields: ' + e);
          throw e; // Can't continue with this script
        }

        $form.validate({
          // Get the hosted form widget for the submitted form (Form1 only)
          submitHandler: function () {
            try {
              console.log('getPaymentToken');

              // Check we have some enabled fields to submit
              if ($($form[0].elements).filter(':enabled:not([type="hidden"])').length === 0) {
                showError('You must enable some fields');
                return false;
              }

              var hostedform = $form.hostedForm('instance');

              var also = {
                customerName: $('#form-customer-name').val()
              };

              hostedform.getPaymentDetails(also, true).then(

                // Success validating the form and requesting a payment token
                function (details) {
                  if (details.success) {
                    $form[0].elements['paymentToken'].value = details.paymentToken;
                    $form[0].submit();
                  } else if (details.invalid) {
                    $form.valid();
                    showFieldErrors(details.invalid);
                  } else {
                    showError('There was a problem fetching the payment token. Please seek assistance.');
                  }
                },

                // Failure either validating the form or requesting the payment details
                function (e) {
                  showError('There was a problem fetching the payment token. Please seek assistance.');
                }
              );

            } catch (e) {
              showError('There was a problem fetching the payment token. Please seek assistance.');
            }
          }
        });

        // End of form manual setup

      }
```

```

        // Hide errors once all fields are valid
        $('#form :input').on('valid', function () {
          if ($(this.form).find(':invalid').length === 0) {
            hideError($(this.form));
          }
        })

        // Listen for some events on the none Hosted Fields
        $('.form-control-native').on('invalid', bsMarkInvalid);
        $('.form-control-native').on('valid', bsMarkValid);

        // Check we can see the Hosted Fields via their new class
        // (For demonstration purposes only)
        console.log($('.form-control-hosted.hostedfield-element'));

        // Check we can see the Hosted Fields via the psuedo element
        // (For demonstration purposes only)
        console.log($('.form-control:hostedfield'));

      });

      ///////////////////////////////////////////////////////////////////
      // Supporting functions
      ///////////////////////////////////////////////////////////////////

      // Display events that are passed from hosted field
      var events = {
        'hostedfield:create.example'                              : showEvent,
        'hostedfield:destroy.example'                             : showEvent,
        'hostedfield:ready.example'                               : showEvent,
        'hostedfield:style.example'                               : showEvent,
        'hostedfield:placeholder.example'                         : showEvent,
        'hostedfield:invalid.example invalid.example'             : showEvent,
        'hostedfield:userinvalid.example userinvalid.example'     : showEvent,
        'hostedfield:valid.example valid.example'                 : showEvent,
        'hostedfield:uservalid.example uservalid.example'         : showEvent,
        'hostedfield:disabled.example disabled.example'           : showEvent,
        'hostedfield:enabled.example enabled.example'             : showEvent,
        'hostedfield:required.example required.example'           : showEvent,
        'hostedfield:optional.example optional.example'           : showEvent,
        'hostedfield:readonly.example readonly.example'           : showEvent,
        'hostedfield:readwrite.example readwrite.example'         : showEvent,
        'hostedfield:focus.example focus.example'                 : showEvent,
        'hostedfield:blur.example blur.example'                   : showEvent,
        'hostedfield:mouseenter.example mouseenter.example'       : showEvent,
        'hostedfield:mouseleave.example mouseleave.example'       : showEvent,
        'hostedfield:mouseover.example mouseover.example'         : showEvent,
        'hostedfield:mouseout.example mouseout.example'           : showEvent,
        'hostedfield:mousemove.example mousemove.example'         : showEvent,
        'hostedfield:keydown.example keydown.example'             : showEvent,
        'hostedfield:keypress.example keypress.example'           : showEvent,
        'hostedfield:keyup.example keyup.example'                 : showEvent,
        'hostedfield:change.example change.example'               : showEvent,
        'hostedfield:input.example input.example'                 : showEvent,

        'hostedfield:invalid.example invalid.example'             : bsMarkInvalid,
        'hostedfield:valid.example valid.example'                 : bsMarkValid,
        'hostedfield:valid.example valid.example'                 : hideError,
      };

      function isInvalid(element) {
        return !element[0].checkValidity();
      }

      function showError(msg) {
        $('#error-info').html(msg).show();
      }

      function hideError($form, msg) {
        $('#error-info', $form).hide();
```

```
    }

    function showFieldErrors(errors) {
      var msg = '<h5>Error</h5><p>The following fields are invalid:</p><ul>';
      for (var p in errors) {
        msg += '<li><b>' + p + ':</b> ' + errors[p] + '</li>';
      }
      msg += '</ul>'
      showError(msg);
    }

    function bsMarkInvalid(e) {
      var element = (e instanceof $.Event ? this : e);
      $(element).closest('.form-group').addClass('has-error');
    }

    function bsMarkValid(e) {
      var element = (e instanceof $.Event ? this : e);
      $(element).closest('.form-group').removeClass('has-error');
    }

    function showEvent(event) {
      console.log(event);
      console.log('Field ' + event.type + ' event: ', this, arguments);
    }

    jQuery.validator.setDefaults({
      ignore: [],
      rules: {
        'customer-name': {
          checkValidity: true,
          required: false
        },
        'card-details': {
          checkValidity: true,
          required: false
        },
        'card-number': {
          checkValidity: true,
          required: false
        },
        'card-expiry-date': {
          checkValidity: true,
          required: false
        },
        'card-start-date': {
          checkValidity: true,
          required: false
        },
        'card-issue-number': {
          checkValidity: true,
          required: false
        },
        'card-cvv': {
          checkValidity: true,
          required: false
        }
      },
      keyup: null, // Don\'t validate on keyup
      showErrors: function (errorMap, errorList) {
        if (errorList && errorList.length) {
          var errors = {};
          for (var i = 0, max_i = errorList.length; i < max_i; i++ ) {
            var label = $('label[for="' + errorList[i].element.id + '"]:not(".error")').text();
            errors[label] = errorList[i].message;
          }
          showFieldErrors(errors);
        }
        this.defaultShowErrors(errorMap, errorList);
      },
      highlight: bsMarkInvalid,
        unhighlight: bsMarkValid,
        errorPlacement: function (error, element) {
          $(element).closest('.form-control:not(".hostedfield-element")').after(error);
        }
      });

      $.validator.addMethod('checkValidity',
        function (value, element, params, message) {
          element.checkValidity();
          var valid = (element.validationMessage === '');
          $(element).attr('aria-invalid', !valid);
          return valid;
        },
        function (params, element) {
          return element.validationMessage;
        }
      );

    </script>

  </body>
</html>
```