To integrate the Tokenizer API into your React website application with a frontend and backend, you’ll need to set up the Tokenizer JavaScript library on the frontend to collect and tokenize payment information securely, then use the resulting token in your backend to process transactions via FluidPay’s API. Below is a step-by-step guide based on the provided documentation and the sandbox link at https://sandbox.fluidpay.com/docs/services/tokenizer.

---

### Prerequisites

1. \*\*FluidPay Account\*\*: Ensure you have a FluidPay account with access to a \*\*public API key\*\* (starts with `pub\_`) for the frontend and a \*\*secret API key\*\* for the backend.

2. \*\*React Application\*\*: A working React frontend and a backend server (e.g., Node.js, Express, or another framework).

3. \*\*Sandbox Environment\*\*: Use the sandbox endpoint (`https://sandbox.fluidpay.com`) for testing.

---

### Step 1: Set Up the Frontend (React)

The Tokenizer library injects an iframe to collect sensitive payment data (e.g., credit card or ACH details) without bringing your website into PCI compliance scope. Here’s how to integrate it into your React application.

#### 1.1. Add the Tokenizer Script

You need to include the Tokenizer JavaScript library in your React app. Since the documentation suggests using a `<script>` tag, you can load it dynamically or include it globally.

\*\*Option 1: Load via `<script>` Tag\*\*

Add the Tokenizer script to your `public/index.html` file:

```html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8" />

<title>React App</title>

<!-- Add Tokenizer script -->

<script src="https://sandbox.fluidpay.com/static/tokenizer/1.0.0/tokenizer.js"></script>

</head>

<body>

<div id="root"></div>

</body>

</html>

```

\*\*Option 2: Import Dynamically (for Single Page Apps)\*\*

If you’re using a module bundler like Webpack, you can import the script dynamically in your React component to ensure compatibility with Single Page Applications (SPAs):

```javascript

import { useEffect } from 'react';

const loadTokenizerScript = () => {

return new Promise((resolve, reject) => {

const script = document.createElement('script');

script.src = 'https://sandbox.fluidpay.com/static/tokenizer/1.0.0/tokenizer.js';

script.async = true;

script.onload = () => resolve(window.Tokenizer);

script.onerror = () => reject(new Error('Failed to load Tokenizer script'));

document.body.appendChild(script);

});

};

```

#### 1.2. Create a Payment Form Component

Create a React component to render the container for the Tokenizer iframe and handle form submission.

```javascript

import { useEffect, useRef } from 'react';

function PaymentForm() {

const containerRef = useRef(null);

const tokenizerRef = useRef(null);

useEffect(() => {

// Load the Tokenizer script

loadTokenizerScript().then(() => {

// Initialize Tokenizer

tokenizerRef.current = new window.Tokenizer({

apikey: 'pub\_XXXXXXXXXXXXXX', // Replace with your public API key

container: containerRef.current, // Reference to the container div

submission: (resp) => {

console.log('Submission response:', resp);

if (resp.status === 'success') {

// Send token to backend for processing

handleToken(resp.token);

} else if (resp.status === 'error') {

console.error('Error:', resp.msg);

} else if (resp.status === 'validation') {

console.error('Validation errors:', resp.invalid);

}

},

onLoad: () => console.log('Tokenizer iframe loaded'),

settings: {

payment: {

types: ['card', 'ach'], // Enable both card and ACH

calculateFees: true, // Optional: calculate fees

card: {

requireCVV: true, // Require CVV for cards

strict\_mode: false, // Allow 19-digit cards

},

ach: {

sec\_code: 'web', // Default ACH SEC code

showSecCode: false, // Hide SEC code dropdown

verifyAccountRouting: false, // Disable account/routing verification

},

},

user: {

showInline: true, // Show user fields (name, email, etc.)

showName: true,

showEmail: true,

showPhone: true,

},

billing: {

show: true, // Show billing address fields

showTitle: true,

},

styles: {

// Optional: Customize iframe styles

'input': {

'color': '#ffffff',

'border-radius': '8px',

'background-color': '#ffffff40',

'border': 'none',

},

},

},

});

}).catch((error) => console.error(error));

// Cleanup on component unmount

return () => {

if (tokenizerRef.current) {

// Note: Tokenizer may not have a destroy method; ensure proper cleanup if needed

}

};

}, []);

// Function to handle token submission to backend

const handleToken = async (token) => {

try {

const response = await fetch('http://your-backend-api/process-payment', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ token }),

});

const result = await response.json();

console.log('Backend response:', result);

} catch (error) {

console.error('Error sending token to backend:', error);

}

};

// Trigger form submission

const handleSubmit = () => {

if (tokenizerRef.current) {

tokenizerRef.current.submit(); // Optionally pass amount: tokenizerRef.current.submit("5.55")

}

};

return (

<div>

<div id="container" ref={containerRef}></div>

<button onClick={handleSubmit}>Submit Payment</button>

</div>

);

}

export default PaymentForm;

```

\*\*Key Points\*\*:

- \*\*Container\*\*: The `containerRef` points to a `<div>` where the Tokenizer iframe will be injected.

- \*\*Public API Key\*\*: Replace `'pub\_XXXXXXXXXXXXXX'` with your actual public API key from FluidPay.

- \*\*Submission Callback\*\*: The `submission` callback handles the response (`resp.token` for success, errors, or validation issues).

- \*\*Styling\*\*: Customize the iframe’s appearance using the `styles` object in `settings`.

- \*\*ACH/Card Support\*\*: Enable both payment types by setting `types: ['card', 'ach']`.

- \*\*Sandbox URL\*\*: Ensure the script points to `https://sandbox.fluidpay.com/static/tokenizer/1.0.0/tokenizer.js` for testing.

#### 1.3. Handle SPA Considerations

Since React is a Single Page Application framework, the documentation notes that Tokenizer is set up for UMD (Universal Module Definition). If you encounter issues with Webpack or module bundling, use the dynamic import method above or ensure the script is globally available via `window.Tokenizer`.

---

### Step 2: Set Up the Backend

The backend will use the token generated by the Tokenizer to process payments via FluidPay’s Transaction Processing API. The token replaces sensitive payment data in API calls and expires after 2 minutes.

#### 2.1. Backend API Endpoint

Create an endpoint to receive the token from the frontend and make a request to FluidPay’s Transaction Processing API.

\*\*Example (Node.js with Express)\*\*:

```javascript

const express = require('express');

const axios = require('axios');

const app = express();

app.use(express.json());

// Endpoint to process payment

app.post('/process-payment', async (req, res) => {

const { token } = req.body;

try {

const response = await axios.post(

'https://sandbox.fluidpay.com/api/transaction/sale',

{

payment\_method: {

token: token, // Use the token from the frontend

},

amount: 555, // Amount in cents (e.g., $5.55)

currency: 'USD',

},

{

headers: {

Authorization: 'Bearer sk\_XXXXXXXXXXXXXX', // Replace with your secret API key

'Content-Type': 'application/json',

},

}

);

res.json({

status: 'success',

data: response.data,

});

} catch (error) {

console.error('Error processing payment:', error.response?.data || error.message);

res.status(500).json({

status: 'error',

message: error.response?.data?.message || 'Payment processing failed',

});

}

});

app.listen(3000, () => console.log('Server running on port 3000'));

```

\*\*Key Points\*\*:

- \*\*Secret API Key\*\*: Use your FluidPay secret API key (`sk\_XXXXXXXXXXXXXX`) for backend requests. Never expose this key on the frontend.

- \*\*Token Usage\*\*: Pass the token in the `payment\_method.token` field of the Transaction Processing API request.

- \*\*Endpoint\*\*: Use `https://sandbox.fluidpay.com/api/transaction/sale` for testing sales. See the [FluidPay API documentation](https://sandbox.fluidpay.com/docs/api) for other endpoints (e.g., `/authorize`, `/refund`).

- \*\*Amount\*\*: Ensure the amount is in cents (e.g., `$5.55` = `555`).

#### 2.2. Security Considerations

- \*\*Secret API Key\*\*: Store the secret API key securely (e.g., in environment variables using `dotenv`).

- \*\*CORS\*\*: Configure CORS on your backend to allow requests from your frontend domain.

- \*\*Token Expiration\*\*: Tokens expire after 2 minutes, so process them immediately after receiving them from the frontend.

---

### Step 3: Testing in the Sandbox

1. \*\*Obtain API Keys\*\*:

- Get your public (`pub\_`) and secret (`sk\_`) API keys from the FluidPay sandbox dashboard.

2. \*\*Test Payment\*\*:

- Use test card numbers or ACH details from FluidPay’s [sandbox documentation](https://sandbox.fluidpay.com/docs/api/test-data).

- Example test card: `4111111111111111`, Exp: `12/25`, CVV: `123`.

3. \*\*Verify Tokenization\*\*:

- Ensure the Tokenizer iframe loads in your React app and returns a token in the `submission` callback.

4. \*\*Verify Transaction\*\*:

- Check that your backend successfully processes the token with FluidPay’s API and returns a response.

---

### Step 4: Optional Features

Based on the documentation, you can enhance the integration with these features:

#### 4.1. Styling the Iframe

Customize the iframe’s appearance using the `settings.styles` object (as shown in the React component example). Inspect the iframe’s DOM in your browser’s developer tools to identify CSS selectors.

#### 4.2. Enable ACH Payments

Include `'ach'` in `payment.types` and configure ACH-specific settings like `sec\_code` or `verifyAccountRouting` (see the React component example).

#### 4.3. Fee Calculation

Enable `calculateFees: true` in `settings.payment` to calculate fees based on the amount and processor. Pass the `processorId` if you have a specific processor configured.

#### 4.4. Paay Integration (3D Secure)

If using Paay for 3D Secure authentication, enable it in the Tokenizer settings and pass the amount to `tokenizer.submit(amount)`:

```javascript

settings: {

paay: {

sandbox: true, // Use Paay sandbox

forceDisabled: false,

rejectChallenges: [], // Define statuses to reject

},

}

```

Then, handle Paay responses in the `submission` callback (`resp.paay`).

#### 4.5. Card Swipe Reader

If you have a card swipe reader, focus on the credit card field in the iframe, and Tokenizer will automatically populate the card number and expiration date.

---

### Step 5: Konnektive Integration (Optional)

If using Konnektive as your CRM, submit the token to Konnektive’s Order Import API in the `formCardNonce` field. Refer to Konnektive’s documentation for details on their API.

\*\*Example\*\*:

```javascript

const response = await axios.post(

'https://api.konnektive.com/order/import/',

{

formCardNonce: token, // Token from Tokenizer

// Other required Konnektive fields (e.g., orderId, amount)

},

{

headers: {

Authorization: 'Bearer YOUR\_KONNEKTIVE\_API\_KEY',

'Content-Type': 'application/json',

},

}

);

```

---

### Step 6: Go Live

1. \*\*Switch to Production\*\*: Replace `https://sandbox.fluidpay.com` with `https://app.fluidpay.com` in both the Tokenizer script URL and backend API calls.

2. \*\*Update API Keys\*\*: Use production API keys (`pub\_` and `sk\_`) from your FluidPay account.

3. \*\*Test Thoroughly\*\*: Ensure all payment flows (card, ACH, Paay, etc.) work in the sandbox before going live.

4. \*\*PCI Compliance\*\*: Since Tokenizer handles sensitive data in an iframe, your website remains out of PCI scope, but ensure your backend securely handles tokens.

---

### Troubleshooting

- \*\*Tokenizer Not Loading\*\*: Ensure the script URL is correct and the container element exists when the Tokenizer is initialized. Use `onLoad` to verify iframe loading.

- \*\*Token Errors\*\*: Check for `error` or `validation` statuses in the `submission` callback and log `resp.msg` or `resp.invalid`.

- \*\*Backend Errors\*\*: Verify your secret API key and ensure the token is sent within 2 minutes.

- \*\*CORS Issues\*\*: Configure your backend to allow requests from your frontend’s domain.

- \*\*Local Development\*\*: Set the API endpoint explicitly if developing locally (e.g., `https://sandbox.fluidpay.com`).

---

### Example Workflow

1. User loads your React app, and the PaymentForm component renders the Tokenizer iframe.

2. User enters payment details (card or ACH) and clicks “Submit.”

3. Tokenizer returns a token in the `submission` callback.

4. The frontend sends the token to your backend.

5. The backend uses the token to process a transaction via FluidPay’s API.

6. The backend returns the result (success or error) to the frontend for display.

---

### Additional Notes

- \*\*SuperGrok Limitations\*\*: As Grok 3, I cannot access pricing details for SuperGrok or x.com premium subscriptions. For SuperGrok pricing, visit https://x.ai/grok. For x.com subscriptions, visit https://help.x.com/en/using-x/x-premium.

- \*\*API Service\*\*: For more details on FluidPay’s API, refer to https://x.ai/api (though this may be unrelated to FluidPay; use FluidPay’s official docs for accuracy).

- \*\*Documentation Reference\*\*: The provided document and https://sandbox.fluidpay.com/docs/services/tokenizer were used to ensure accuracy.

If you need help with specific code modifications, debugging, or additional features (e.g., Paay, fee calculation), let me know!

To integrate the Tokenizer API into your React website application for \*\*production\*\* credit card processing (no sandbox), you’ll need to set up the Tokenizer JavaScript library on the frontend to securely collect and tokenize payment information, then process the token on your backend using FluidPay’s production API. Below is a comprehensive guide tailored for production, based on the provided documentation and adjusted for the production environment (https://app.fluidpay.com).

---

### Prerequisites

1. \*\*FluidPay Account\*\*: You must have a live FluidPay account with a \*\*public API key\*\* (starts with `pub\_`) for the frontend and a \*\*secret API key\*\* (starts with `sk\_`) for the backend. Obtain these from your FluidPay dashboard.

2. \*\*React Application\*\*: A working React frontend and a backend server (e.g., Node.js with Express or another framework).

3. \*\*PCI Compliance\*\*: Since Tokenizer uses an iframe to handle sensitive payment data, your website remains out of PCI scope, but ensure your backend securely handles tokens.

4. \*\*HTTPS\*\*: Your website must use HTTPS in production to ensure secure communication.

---

### Step 1: Set Up the Frontend (React)

The Tokenizer library injects a secure iframe to collect credit card details, tokenizing them without exposing sensitive data to your application. Here’s how to integrate it into your React app for production.

#### 1.1. Add the Tokenizer Script

In production, use the production Tokenizer script URL: `https://app.fluidpay.com/static/tokenizer/1.0.0/tokenizer.js`. You can include it in your `public/index.html` or load it dynamically for Single Page Applications (SPAs).

\*\*Option 1: Add to `public/index.html`\*\*

Modify your `public/index.html` to include the production Tokenizer script:

<xaiArtifact artifact\_id="f2585b83-ecc2-424d-b2f1-dcaa5034ddb6" artifact\_version\_id="cfa5318c-5f84-4e63-b004-f84298d5e6a0" title="index.html" contentType="text/html">

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8" />

<title>React Payment App</title>

<!-- Production Tokenizer script -->

<script src="https://app.fluidpay.com/static/tokenizer/1.0.0/tokenizer.js"></script>

</head>

<body>

<div id="root"></div>

</body>

</html>

</xaiArtifact>

\*\*Option 2: Dynamic Import (for SPAs)\*\*

For better control in React SPAs, load the script dynamically in your component:

<xaiArtifact artifact\_id="29e3eff4-ce8d-4e1e-af45-55306c72f861" artifact\_version\_id="3522340d-b243-4e0b-a9fa-0f77707c750d" title="PaymentForm.jsx" contentType="text/javascript">

import { useEffect, useRef } from 'react';

const loadTokenizerScript = () => {

return new Promise((resolve, reject) => {

const script = document.createElement('script');

script.src = 'https://app.fluidpay.com/static/tokenizer/1.0.0/tokenizer.js';

script.async = true;

script.onload = () => resolve(window.Tokenizer);

script.onerror = () => reject(new Error('Failed to load Tokenizer script'));

document.body.appendChild(script);

});

};

function PaymentForm() {

const containerRef = useRef(null);

const tokenizerRef = useRef(null);

useEffect(() => {

loadTokenizerScript()

.then(() => {

tokenizerRef.current = new window.Tokenizer({

apikey: 'pub\_XXXXXXXXXXXXXX', // Replace with your production public API key

container: containerRef.current,

submission: (resp) => {

console.log('Submission response:', resp);

if (resp.status === 'success') {

handleToken(resp.token); // Send token to backend

} else if (resp.status === 'error') {

console.error('Error:', resp.msg);

alert('Payment error: ' + resp.msg);

} else if (resp.status === 'validation') {

console.error('Validation errors:', resp.invalid);

alert('Validation error: ' + JSON.stringify(resp.invalid));

}

},

onLoad: () => console.log('Tokenizer iframe loaded'),

onPaymentChange: (type) => console.log('Payment type changed:', type),

validCard: (card) => {

console.log('Valid card input:', card);

if (card.isValid) {

console.log('Card BIN data:', card.bin);

}

},

settings: {

payment: {

types: ['card'], // Credit card only for production

calculateFees: true, // Calculate fees

card: {

requireCVV: true, // Require CVV for security

strict\_mode: false, // Allow 19-digit cards

user\_required: {

first\_name: true,

last\_name: true,

email: true,

phone: false,

},

billing\_required: {

address: true,

city: true,

state: true,

zip: true,

country: true,

},

},

},

user: {

showInline: true,

showName: true,

showEmail: true,

showPhone: true,

showTitle: true,

},

billing: {

show: true,

showTitle: true,

},

styles: {

'body': {

'color': '#333333',

'font-family': 'Arial, sans-serif',

},

'input': {

'color': '#333333',

'border-radius': '4px',

'border': '1px solid #cccccc',

'padding': '8px',

},

'.payment .cvv input': {

'border': '1px solid #cccccc',

'padding-left': '6px',

},

},

},

});

})

.catch((error) => console.error('Script load error:', error));

return () => {

// Cleanup (Tokenizer may not support destroy; handle if needed)

if (tokenizerRef.current) {

console.log('Cleaning up Tokenizer');

}

};

}, []);

const handleToken = async (token) => {

try {

const response = await fetch('https://your-backend-api.com/process-payment', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ token, amount: 1000 }), // Example: $10.00

});

const result = await response.json();

if (result.status === 'success') {

alert('Payment processed successfully!');

} else {

alert('Payment failed: ' + result.message);

}

} catch (error) {

console.error('Error sending token to backend:', error);

alert('Error processing payment');

}

};

const handleSubmit = () => {

if (tokenizerRef.current) {

tokenizerRef.current.submit('10.00'); // Submit with amount in dollars

}

};

return (

<div style={{ padding: '20px' }}>

<h2>Enter Payment Details</h2>

<div id="container" ref={containerRef}></div>

<button

onClick={handleSubmit}

style={{

marginTop: '10px',

padding: '10px 20px',

backgroundColor: '#007bff',

color: '#ffffff',

border: 'none',

borderRadius: '4px',

cursor: 'pointer',

}}

>

Submit Payment

</button>

</div>

);

}

export default PaymentForm;

</xaiArtifact>

\*\*Key Points\*\*:

- \*\*Production Script\*\*: Use `https://app.fluidpay.com/static/tokenizer/1.0.0/tokenizer.js`.

- \*\*Public API Key\*\*: Replace `'pub\_XXXXXXXXXXXXXX'` with your production public API key from FluidPay.

- \*\*Container\*\*: The `containerRef` points to a `<div>` where the iframe is injected.

- \*\*Submission Callback\*\*: Handles success (`resp.token`), errors (`resp.msg`), or validation issues (`resp.invalid`).

- \*\*Settings\*\*: Configured for credit cards only (`types: ['card']`), with required CVV and billing fields for production security.

- \*\*Styling\*\*: Applied basic styles for a clean, professional look. Adjust as needed.

- \*\*Amount\*\*: Pass the amount (e.g., `'10.00'`) to `submit()` for fee calculation or 3D Secure (if enabled).

---

### Step 2: Set Up the Backend

The backend processes the token received from the frontend using FluidPay’s production Transaction Processing API. Tokens expire after 2 minutes, so process them immediately.

#### 2.1. Backend API Endpoint

Create an endpoint to handle the token and make a sale request to FluidPay’s production API.

<xaiArtifact artifact\_id="53165c1b-e48b-4360-a4f9-2d57c356dfd0" artifact\_version\_id="35a8c703-ffdb-4bf5-b657-f1a7fc748040" title="server.js" contentType="text/javascript">

const express = require('express');

const axios = require('axios');

const cors = require('cors');

const app = express();

app.use(cors({ origin: 'https://your-frontend-domain.com' })); // Allow frontend domain

app.use(express.json());

// Environment variables for security

const FLUIDPAY\_SECRET\_KEY = process.env.FLUIDPAY\_SECRET\_KEY; // Set in .env file

app.post('/process-payment', async (req, res) => {

const { token, amount } = req.body;

if (!token || !amount) {

return res.status(400).json({ status: 'error', message: 'Token and amount are required' });

}

try {

const response = await axios.post(

'https://app.fluidpay.com/api/transaction/sale',

{

payment\_method: {

token: token,

},

amount: Math.round(parseFloat(amount) \* 100), // Convert dollars to cents (e.g., 10.00 -> 1000)

currency: 'USD',

billing\_address: {

// Optional: Include if collected by Tokenizer

first\_name: req.body.first\_name || '',

last\_name: req.body.last\_name || '',

address\_line\_1: req.body.address || '',

city: req.body.city || '',

state: req.body.state || '',

postal\_code: req.body.zip || '',

country: req.body.country || 'US',

},

},

{

headers: {

Authorization: `Bearer ${FLUIDPAY\_SECRET\_KEY}`, // Production secret API key

'Content-Type': 'application/json',

},

}

);

res.json({

status: 'success',

data: response.data,

transaction\_id: response.data.data.id,

});

} catch (error) {

console.error('Payment error:', error.response?.data || error.message);

res.status(500).json({

status: 'error',

message: error.response?.data?.message || 'Failed to process payment',

});

}

});

app.listen(3000, () => console.log('Server running on port 3000'));

</xaiArtifact>

\*\*Key Points\*\*:

- \*\*Production Endpoint\*\*: Use `https://app.fluidpay.com/api/transaction/sale` for processing sales.

- \*\*Secret API Key\*\*: Store your production secret API key (`sk\_XXXXXXXXXXXXXX`) in an environment variable (e.g., using `dotenv`).

- \*\*Amount\*\*: Convert the amount from dollars to cents (e.g., `$10.00` → `1000`).

- \*\*CORS\*\*: Configure CORS to allow requests from your frontend domain (e.g., `https://your-frontend-domain.com`).

- \*\*Billing Address\*\*: Include billing fields if collected by Tokenizer and required by your settings.

- \*\*Error Handling\*\*: Return meaningful error messages to the frontend for user feedback.

#### 2.2. Security Considerations

- \*\*Secret API Key\*\*: Never expose the secret API key in your frontend or version control. Use environment variables (e.g., `.env` file).

- \*\*HTTPS\*\*: Ensure your backend API uses HTTPS in production.

- \*\*Token Expiration\*\*: Process tokens within 2 minutes to avoid expiration.

- \*\*Input Validation\*\*: Validate `token` and `amount` to prevent invalid requests.

---

### Step 3: Production Configuration

1. \*\*Obtain Production API Keys\*\*:

- Get your production public (`pub\_`) and secret (`sk\_`) API keys from your FluidPay dashboard.

2. \*\*Update URLs\*\*:

- Frontend: Use `https://app.fluidpay.com/static/tokenizer/1.0.0/tokenizer.js`.

- Backend: Use `https://app.fluidpay.com/api/transaction/sale`.

3. \*\*Required Fields\*\*:

- Enable `requireCVV: true` and required billing/user fields in `settings` to enhance security.

4. \*\*Logging\*\*: Implement secure logging for errors and transactions (e.g., using a logging service) without storing sensitive data.

5. \*\*Monitoring\*\*: Set up monitoring for failed transactions and API errors in production.

---

### Step 4: Optional Features

Enhance the integration with these production-ready features:

#### 4.1. Styling the Iframe

Customize the iframe’s appearance to match your website’s branding using the `settings.styles` object (see the `PaymentForm.jsx` example). Test styles thoroughly to ensure consistency across browsers.

#### 4.2. Fee Calculation

Enable `calculateFees: true` to calculate service fees or surcharges. Optionally specify a `processorId`:

```javascript

settings: {

payment: {

calculateFees: true,

processorId: 'YOUR\_PROCESSOR\_ID', // Optional

},

}

```

Check the `validCard` callback for fee details (`card.ServiceFee`, `card.Surcharge`, etc.).

#### 4.3. 3D Secure (Paay) Integration

For enhanced security, enable Paay for 3D Secure authentication:

<xaiArtifact artifact\_id="cb6a70fd-2b3b-4f50-a5a8-e9928a5101ba" artifact\_version\_id="4f227c2e-b03b-47d9-ac51-baddd9eae21d" title="PaymentForm.jsx" contentType="text/javascript">

import { useEffect, useRef } from 'react';

const loadTokenizerScript = () => {

return new Promise((resolve, reject) => {

const script = document.createElement('script');

script.src = 'https://app.fluidpay.com/static/tokenizer/1.0.0/tokenizer.js';

script.async = true;

script.onload = () => resolve(window.Tokenizer);

script.onerror = () => reject(new Error('Failed to load Tokenizer script'));

document.body.appendChild(script);

});

};

function PaymentForm() {

const containerRef = useRef(null);

const tokenizerRef = useRef(null);

useEffect(() => {

loadTokenizerScript()

.then(() => {

tokenizerRef.current = new window.Tokenizer({

apikey: 'pub\_XXXXXXXXXXXXXX', // Production public API key

container: containerRef.current,

submission: (resp) => {

console.log('Submission response:', resp);

if (resp.status === 'success') {

console.log('Paay response:', resp.paay); // Log Paay details

handleToken(resp.token);

} else if (resp.status === 'error') {

console.error('Error:', resp.msg);

alert('Payment error: ' + resp.msg);

} else if (resp.status === 'validation') {

console.error('Validation errors:', resp.invalid);

alert('Validation error: ' + JSON.stringify(resp.invalid));

}

},

onLoad: () => console.log('Tokenizer iframe loaded'),

settings: {

payment: {

types: ['card'],

calculateFees: true,

card: {

requireCVV: true,

strict\_mode: false,

user\_required: {

first\_name: true,

last\_name: true,

email: true,

phone: false,

},

billing\_required: {

address: true,

city: true,

state: true,

zip: true,

country: true,

},

},

},

paay: {

sandbox: false, // Production mode

forceDisabled: false, // Enable Paay

rejectChallenges: ['challenge\_required'], // Reject specific 3DS challenges

},

user: {

showInline: true,

showName: true,

showEmail: true,

showPhone: true,

showTitle: true,

},

billing: {

show: true,

showTitle: true,

},

styles: {

'body': {

'color': '#333333',

'font-family': 'Arial, sans-serif',

},

'input': {

'color': '#333333',

'border-radius': '4px',

'border': '1px solid #cccccc',

'padding': '8px',

},

},

},

});

})

.catch((error) => console.error('Script load error:', error));

return () => {

if (tokenizerRef.current) {

console.log('Cleaning up Tokenizer');

}

};

}, []);

const handleToken = async (token) => {

try {

const response = await fetch('https://your-backend-api.com/process-payment', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ token, amount: 10.00 }),

});

const result = await response.json();

if (result.status === 'success') {

alert('Payment processed successfully!');

} else {

alert('Payment failed: ' + result.message);

}

} catch (error) {

console.error('Error sending token to backend:', error);

alert('Error processing payment');

}

};

const handleSubmit = () => {

if (tokenizerRef.current) {

tokenizerRef.current.submit('10.00'); // Pass amount for Paay

}

};

return (

<div style={{ padding: '20px' }}>

<h2>Enter Payment Details</h2>

<div id="container" ref={containerRef}></div>

<button

onClick={handleSubmit}

style={{

marginTop: '10px',

padding: '10px 20px',

backgroundColor: '#007bff',

color: '#ffffff',

border: 'none',

borderRadius: '4px',

cursor: 'pointer',

}}

>

Submit Payment

</button>

</div>

);

}

export default PaymentForm;

</xaiArtifact>

\*\*Paay Notes\*\*:

- Set `paay.sandbox: false` for production.

- Pass the amount to `submit()` for Paay to process 3D Secure authentication.

- Check `resp.paay` in the `submission` callback for 3D Secure results.

- Review FluidPay’s [3D Secure documentation](https://docs.3dsintegrator.com/docs/3ds-response-table-of-content) for handling `rejectChallenges`.

#### 4.4. Card Swipe Reader

If using a card swipe reader, ensure the credit card field is focused in the iframe. Tokenizer will automatically populate the card number and expiration date.

---

### Step 5: Konnektive Integration (Optional)

If using Konnektive as your CRM, submit the token to Konnektive’s Order Import API in the `formCardNonce` field.

<xaiArtifact artifact\_id="8bc59fc3-aa21-4c31-a934-afcba437d2cc" artifact\_version\_id="01de9644-b150-4475-99e8-2110d6a0f795" title="konnektive.js" contentType="text/javascript">

const axios = require('axios');

async function submitToKonnektive(token, amount, userData, billingData) {

try {

const response = await axios.post(

'https://api.konnektive.com/order/import/',

{

formCardNonce: token, // Token from Tokenizer

amount: amount, // e.g., 10.00

firstName: userData.first\_name || '',

lastName: userData.last\_name || '',

email: userData.email || '',

address1: billingData.address || '',

city: billingData.city || '',

state: billingData.state || '',

postalCode: billingData.zip || '',

country: billingData.country || 'US',

// Add other required Konnektive fields (e.g., orderId, productId)

},

{

headers: {

Authorization: 'Bearer YOUR\_KONNEKTIVE\_API\_KEY', // Replace with Konnektive API key

'Content-Type': 'application/json',

},

}

);

return { status: 'success', data: response.data };

} catch (error) {

console.error('Konnektive error:', error.response?.data || error.message);

throw new Error(error.response?.data?.message || 'Failed to submit to Konnektive');

}

}

module.exports = { submitToKonnektive };

</xaiArtifact>

Integrate this into your backend’s `/process-payment` endpoint if using Konnektive.

---

### Step 6: Production Best Practices

1. \*\*Security\*\*:

- Use HTTPS for all frontend and backend communications.

- Store the secret API key securely in environment variables.

- Implement rate limiting and input validation on your backend.

2. \*\*Error Handling\*\*:

- Display user-friendly error messages based on `resp.msg` or `resp.invalid` (frontend) and API error responses (backend).

- Log errors securely without storing sensitive data.

3. \*\*Monitoring\*\*:

- Set up monitoring for transaction failures, API errors, and token expiration issues.

- Use FluidPay’s dashboard to track transactions.

4. \*\*Testing in Staging\*\*:

- Before going live, test thoroughly in a staging environment with production-like settings (using production URLs and keys).

5. \*\*Documentation\*\*:

- Refer to https://app.fluidpay.com/docs/services/tokenizer for the latest Tokenizer details.

- Check FluidPay’s API docs at https://app.fluidpay.com/docs/api for transaction endpoints.

---

### Example Workflow

1. User loads your React app, and the `PaymentForm` component renders the Tokenizer iframe.

2. User enters credit card details, CVV, and required billing/user info, then clicks “Submit.”

3. Tokenizer returns a token via the `submission` callback.

4. The frontend sends the token and amount to your backend.

5. The backend processes the payment via `https://app.fluidpay.com/api/transaction/sale`.

6. The backend returns the result to the frontend, which displays success or error messages.

---

### Troubleshooting

- \*\*Tokenizer Not Loading\*\*: Verify the script URL (`https://app.fluidpay.com/static/tokenizer/1.0.0/tokenizer.js`) and ensure the container `<div>` exists.

- \*\*Token Errors\*\*: Check `resp.status` in the `submission` callback. Log `resp.msg` or `resp.invalid` for details.

- \*\*Backend Errors\*\*: Ensure the secret API key is correct and the token is processed within 2 minutes.

- \*\*CORS Issues\*\*: Update CORS settings to allow your frontend domain.

- \*\*3D Secure Issues\*\*: If using Paay, verify the `paay` settings and check `resp.paay` for errors.

---

If you need further assistance with specific features (e.g., Paay, custom styling, or Konnektive integration) or encounter issues, please provide more details, and I’ll tailor the solution further!