Accept Stripe Payments with React and Express

JUNE 20, 2017 BY ROBIN WIERUCH - EDIT THIS POST

Accepting payments on your own website can be an issue at some point. Often you'll find yourself avoiding it, because you can easily outsource it to platforms such as Gumroad that deal with it. However, at some point you want to avoid these third-party platforms and introduce your own payment system.

I dealt with the same issue in a React application. After researching the topic, I found out that using Stripe with React could be a solution, and an viable alternative to PayPal, to charge money. Stripe is a payment platform with a well documented API for developers.

In this React tutorial, you will setup a minimal React application where you can charge money with a credit card React Stripe form and a minimal Express server that receives the payment request. The request gets forwared from your payment server to Stripe via its platform API. The tutorial doesn't come with a lot extra on top. It tries to keep the solution to a minimum so that you can adapt it to your needs.

If you are interested in a full blown payment solution with Stripe and PayPal in React, you can also read about the PayPal in React setup in my other article.

STRIPE PLATFORM

Before you start with the implementation, make sure that you signed up for Stripe. You should create an account and be able to access your dashboard. The dashboard gives you an overview of your payments. There are three things to know about it:

- **Modes:** Somewhere you should be able to switch between live and test mode. When in **test mode**, you will only see payments that were done from your test application. Once you go live with your application, you will see real payments in the **live mode**.
- **API keys:** Somewhere you should find your API keys. One API key is the **publishable API key** that is used in your React application to generate a token. The other **secret API key** is used in your server application to charge the money by accessing the Stripe API.
- **Switching Modes:** Be aware that the API keys change when you switch from live to test mode and vice versa. Thus you can use both variants in your development (test mode) and production (live mode) environments of your application. For instance, when you test your application locally, you can charge money and the payment will be displayed in the dashboard. But only when the dashboard is in test mode. Once you go to production with your application, the money will be charged for real from a credit card.

Now, let's begin implementing the frontend application with React and afterward the backend application with Express. The official Stripe library will be used on both sides: In the React application, the Stripe library generates by using the publishable API key a token. Afterward you will forward the token to your backend application. In the backend, the token is used combined with the Stripe library and the secret API key to charge money via the Stripe API. The separation between frontend and backend is mandatory for security reasons.

Before you dive into the frontend application, let's setup the folder structure. First, create your project folder from the command line:

```
mkdir react-express-stripe
cd react-express-stripe
```

In this folder, you will create your *frontend/* and *backend/* folders in the next chapters.

REACT STRIPE FRONTEND

Using create-react-app is the fastest way to get started in React. It bootstraps your ReactJs project with all boilerplate code with zero-configuration from the command line. You can read more about it in the official documentation. Bootstrap your frontend application grom *react-express-stripe/* on the command line:

```
npx create-react-app frontend
cd frontend
```

The next step is to install a couple of libraries:

```
npm install --save axios react-stripe-checkout
```

You will use axios to make your payment request to your own Express backend. However, it is up to you to use another solution such as superagent or the native fetch API of the browser. After all, you are in a React environment, so you can opt-in whatever solution suits you.

The second library you will use is react-stripe-checkout. It does two things for you:

- it comes with a pretty component to capture credit card information
- it generates a Stripe token that you can send afterward to your backend

I guess the token generation happens under the hood with the official Stripe.js library that you will later use in your backend too.

There exists another library, the official library by Stripe for React, called react-stripe-elements. However, I ran into two drawbacks when using it:

- it isn't supported when using server side rendering (e.g. with NextJs)
- it didn't came with a pretty pre-configured Stripe like style

After you have installed all the necessary libraries, the frontend application needs only a handful more folders and files for a couple of constants and one component. From *react-express-stripe/frontend* on the command line:

```
cd src
touch Checkout.js
mkdir constants && cd constants
touch server.js stripe.js
```

Let's start in the *src/App.js* component that comes from create-react-app. Replace it with the following code:

```
import React, { Component } from 'react';
import logo from './logo.svg';
import Checkout from './Checkout';
import './App.css';

class App extends Component {
  render() {
    return (
      <div className="App">
```

```
          <div className="App-header">
            <img src={logo} className="App-logo" alt="logo" />
            <h2>Welcome to React</h2>
          </div>
          <p className="App-intro">
            <Checkout
              name={'The Road to learn React'}
              description={'Only the Book'}
              amount={1}
            />
          </p>
        </div>
      );
    }
  }

  export default App;
```

The important part is the `Checkout` component. The `name`, `description` and amount can be replaced with your own properties. You can make it a dynamic input with your own forms too.

The second step is to implement the `Checkout` component. The file should be already created by you on the command line.

```
import React from 'react'
import axios from 'axios';
import StripeCheckout from 'react-stripe-checkout';

import STRIPE_PUBLISHABLE from './constants/stripe';
import PAYMENT_SERVER_URL from './constants/server';

const CURRENCY = 'EUR';

const fromEuroToCent = amount => amount * 100;

const successPayment = data => {
  alert('Payment Successful');
};

const errorPayment = data => {
  alert('Payment Error');
};

const onToken = (amount, description) => token =>
  axios.post(PAYMENT_SERVER_URL,
    {
      description,
      source: token.id,
      currency: CURRENCY,
      amount: fromEuroToCent(amount)
    })
    .then(successPayment)
    .catch(errorPayment);

const Checkout = ({ name, description, amount }) =>
  <StripeCheckout
    name={name}
```

```
      description={description}
      amount={fromEuroToCent(amount)}
      token={onToken(amount, description)}
      currency={CURRENCY}
      stripeKey={STRIPE_PUBLISHABLE}
    />

export default Checkout;
```

The `Checkout` component uses the `StripeCheckout` that comes from the library you've installed before. This component receives your personal props, such as `name`, `amount` and `description`, but also needs a `currency` and your `stripeKey` which is the publishable API key from the Stripe dashboard.

The most important prop is the `token` handler. There you can pass your callback function that will be triggered when the user submits the credit card information. The component library already creates the `token` for you in this callback function. Hence you can send all the necessary information to your backend. Because a promise is returned in axios, you can branch your success and error functions with then and catch.

The `Checkout` component uses constants from two files that you have already created. The last step for the frontend is to define these constants.

First, in the *src/constants/server.js* file, you define the URL of your server. In production, when using the express default server, it will be `http://localhost:8080`. In production you may have a proper domain for your server.

```
const PAYMENT_SERVER_URL = process.env.NODE_ENV === 'production'
  ? 'http://myapidomain.com'
  : 'http://localhost:8080';

export default PAYMENT_SERVER_URL;
```

Second, in the *src/constants/stripe.js* file, you define your Stripe API keys depending on development or production environment. You can get your own API keys from the Stripe dashboard and replace them with the following constants.

```
const STRIPE_PUBLISHABLE = process.env.NODE_ENV === 'production'
  ? 'pk_live_MY_PUBLISHABLE_KEY'
  : 'pk_test_MY_PUBLISHABLE_KEY';

export default STRIPE_PUBLISHABLE;
```

That's it for the React Stripe frontend application. You can start it with `npm start` from the command line and test the button. On submit it should fail, because you have no server yet.

EXPRESS STRIPE BACKEND

This chapter gives you guidance to implement your Express server application. It will receive the payment information from your React frontend and will pass it to the Stripe API. In the beginning, navigate into the *react-express-stripe/backend/* folder. There you can initialize a plain npm project:

```
npm init -y
```

By using the `-y` flag, you initialize all the defaults for the project. That's fine for the purpose of this project. As in your frontend application, you have to install a couple of libraries in your backend application.

```
npm install --save express cors body-parser stripe
```

You will use express to create a backend app with a RESTful routing. In addition, there are cors and body-parser to configure and modify straight forward your incoming requests to the Express server. Last but not least, Stripe is this time used as the pure library that enables you to communicate with the Stripe platform API.

Next you can bootstrap your file and folder structure:

```
touch index.js server.js
mkdir constants routes
```

In each new folder you create a few more files:

```
cd constants
touch frontend.js server.js stripe.js
cd ..
cd routes
touch index.js payment.js
```

Now, let's implement a simple backend API that your frontend can send the generated token to the backend and the backend can forward it to the Stripe API. You'll begin with the *index.js* file:

```
const express = require('express');

const SERVER_CONFIGS = require('./constants/server');

const configureServer = require('./server');
const configureRoutes = require('./routes');
```

```
const app = express();

configureServer(app);
configureRoutes(app);

app.listen(SERVER_CONFIGS.PORT, error => {
  if (error) throw error;
  console.log('Server running on port: ' + SERVER_CONFIGS.PORT);
});
```

The *index.js* file basically is your entry point and bootstraps your Express application. The app gets created with `express()`, uses a couple of configurations, that you will define later on, and finally listens on a defined port.

The second step is to define your constants in the different files. First, you can start in *constants/frontend.js*:

```
const FRONTEND_DEV_URLS = [ 'http://localhost:3000' ];

const FRONTEND_PROD_URLS = [
  'https://www.yourdomain.com',
  'https://yourdomain.com'
];

module.exports = process.env.NODE_ENV === 'production'
  ? FRONTEND_PROD_URLS
  : FRONTEND_DEV_URLS;
```

These URLs will be used later to create a whitelist for CORS. In our case only the `FRONTEND_DEV_URLS` matters. But when your application goes in production, you should use your own domain of your frontend application.

Second, the *constants/server.js* file:

```
const path = require('path');

const SERVER_PORT = 8080;

const SERVER_CONFIGS = {
  PRODUCTION: process.env.NODE_ENV === 'production',
  PORT: process.env.PORT || SERVER_PORT,
};

module.exports = SERVER_CONFIGS;
```

The configuration is already used in your *index.js* file to start up the app.

Third, you will define the last constants in *constants/stripe.js*. There you will define the Stripe API keys, similar to the frontend, but this time the secret API keys. Just replace them with your own API keys from your Stripe dashboard.

```
const configureStripe = require('stripe');

const STRIPE_SECRET_KEY = process.env.NODE_ENV === 'production'
    ? 'sk_live_MY_SECRET_KEY'
    : 'sk_test_MY_SECRET_KEY';

const stripe = configureStripe(STRIPE_SECRET_KEY);

module.exports = stripe;
```

Now, only the server configuration and the routing is missing. Let's begin with the server configuration in *server.js*.

```
const cors = require('cors');
const bodyParser = require('body-parser');

const CORS_WHITELIST = require('./constants/frontend');

const corsOptions = {
  origin: (origin, callback) =>
    (CORS_WHITELIST.indexOf(origin) !== -1)
      ? callback(null, true)
      : callback(new Error('Not allowed by CORS'))
};

const configureServer = app => {
  app.use(cors(corsOptions));

  app.use(bodyParser.json());
};

module.exports = configureServer;
```

Basically you enable CORS for your application so that your frontend application is able to communicate with your backend application. In addition, you apply the body-parser middleware to parse your incoming requests rather than parsing them yourself. You don't need to bother about this anymore.

Now, last but not least, comes the routing of your backend application where all the magic happens. In your *routes/payment.js* file you can use the following code:

```
const stripe = require('../constants/stripe');

const postStripeCharge = res => (stripeErr, stripeRes) => {
  if (stripeErr) {
    res.status(500).send({ error: stripeErr });
  } else {
    res.status(200).send({ success: stripeRes });
  }
}

const paymentApi = app => {
```

```
  app.get('/', (req, res) => {
    res.send({ message: 'Hello Stripe checkout server!', timestamp: new Dat
  });

  app.post('/', (req, res) => {
    stripe.charges.create(req.body, postStripeCharge(res));
  });

  return app;
};

module.exports = paymentApi;
```

Basically on a post request, that you are already doing with axios from your React frontend application, you will use the Stripe library to create a official Stripe payment. The payment creation receives the incoming payload from your frontend application, all the credit card information and optional information, and a callback function that executes after the request to the Stripe API succeeds or fails. Afterward, you can send back a response to your React frontend application.

Finally you can wire up your Express application with the payment route in *routes/index.js*:

```
const paymentApi = require('./payment');

const configureRoutes = app => {
  paymentApi(app);
};

module.exports = configureRoutes;
```

The configuration is already used in your *index.js*. Your Express Stripe Backend should work now. Start it with `node index.js` from your *backend/* folder on the command line.

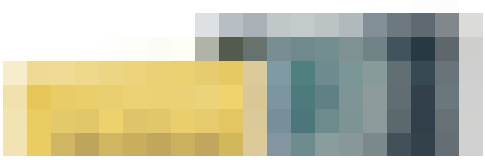
MAKE YOUR FIRST PAYMENT

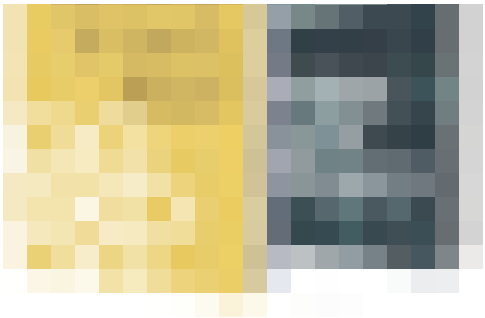
All the implementation is done. Now it is about testing it. When you start backend and frontend, your applications should run on the localhost ports 8080 and 3000. Open up the backend application and verify that it is running on the URL localhost:8080. Open up the frontend application on the URL localhost:3000 and charge money with one of Stripe's test credit cards. One of these credit cards could be:

- Email: Any Email
- Number: 4242 4242 4242 4242
- Date: Any Date in the Future
- CVC: Any 3 Numbers

There should be an obvious alert when the payment succeeded, but also an error alert when it failed. Once it succeeded, you can find the payment on your Stripe dashboard using the test mode. If you charged a real credit card in production environment, the payment should be visible on the dashboard in live mode.

You can find the final application and the installation README.md on GitHub. If you like it, make sure to star it. Otherwise, if you need help to deploy your Stripe payment server to production, follow this Digital Ocean deployment guide. I deploy all my applications there with Dokku.





View our Privacy Policy.

PORTFOLIO

Online Courses

Open Source

Tutorials

ABOUT

About me

What I use

How to work with me

How to support me

Contact Me Privacy & Terms