In this advanced tutorial, you'll learn how to use Gatsby to build the UI for a basic e-commerce site that can accept payments, with Stripe as the backend for processing payments.

- Demo running on Netlify
- Code hosted on GitHub

Why use Gatsby for an E-commerce site?

Benefits of using Gatsby for e-commerce sites include the following:

- Security inherent in static sites.
- Blazing fast performance when your pages are converted from React into static files.
- No server component required with Stripe's client-only Checkout.
- Cost-efficient hosting of static sites.

Prerequisites

- Since this is a more advanced tutorial, building a site with Gatsby before will likely make this tutorial less time-consuming (see the main tutorial here)
- Stripe account: register for an account here

How does Gatsby work with Stripe?

Stripe is a payment processing service that allows you to securely collect and process payment information from your customers. To try out Stripe for yourself, go to Stripe's Quick Start Guide.

Stripe offers a hosted checkout that doesn't require any backend component. You can configure products, prices, and subscription plans in the Stripe Dashboard. If you're selling a single product or subscription (like an eBook) you can hardcode the product's price ID in your Gatsby site. If you're selling multiple products, you can use the Stripe source plugin to retrieve all prices at build time. If you want your Gatsby site to automatically update, you can use the Stripe webhook event to trigger a redeploy when a new product or price is added.

Setting up a Gatsby site

Create a new Gatsby project by running the `gatsby new` command in the terminal and change directories into the new project you just started:

```
gatsby new e-commerce-gatsby-tutorial
cd e-commerce-gatsby-tutorial
```

See your site hot reload in the browser!

Run `gatsby develop` in the terminal, which starts a development server and reloads changes you make to your site so you can preview them in the browser. Open up your browser to `http://localhost:8000/` and you should see a default homepage.

Loading Stripe.js

Stripe provides a JavaScript library that allows you to securely redirect your customer to the Stripe hosted checkout page. Due to PCI compliance requirements, the Stripe.js library has to be loaded from Stripe's servers. Stripe provides a loading wrapper that allows you to import Stripe.js as an ES module. To improve your site's performance, you can hold off instantiating Stripe until your user hits the checkout button. `<YOUR STRIPE PUBLISHABLE KEY>` must be replaced with your own Stripe key.

```
import { loadStripe } from "@stripe/stripe-js"

let stripePromise
const getStripe = () => {
  if (!stripePromise) {
    stripePromise = loadStripe("<YOUR STRIPE PUBLISHABLE KEY>")
  }
  return stripePromise
}
```

Stripe.js is loaded as a side effect of the `import '@stripe/stripe-js';` statement. To best leverage Stripe's advanced fraud functionality, ensure that Stripe.js is loaded on every page of your customer's checkout journey, not just your checkout page. This allows Stripe to detect anomalous behavior that may be indicative of fraud as customers browse your website.

To make use of this, install the `stripe-js` module:

```
npm install @stripe/stripe-js
```

Getting your Stripe test keys

View your API credentials by logging into your Stripe account, and then going to Developers > API Keys.

Stripe public test key location in Stripe account

You have 2 keys in both test mode and production mode:

- a publishable key
- a secret key

While testing, you must use the key(s) that include *test*. For production code, you will need to use the live keys. As the names imply, your publishable key may be included in code that you share publicly (for example, on the frontend, and in GitHub), whereas your secret key should not be shared with anyone or committed to any public repo. It's important to restrict access to this secret key because anyone who has it could potentially read or send requests from your Stripe account and see information about charges or purchases or even refund customers.

Enabling the "Checkout client-only integration" for your Stripe account

In this tutorial you will be using Stripe Checkout in client-only mode. You need to enable client-only mode in the Checkout settings.

Stripe control to enable the Checkout client-side only integration highlighted

> 💡 *This change will also modify the interface that Stripe provides to administer your products: keep this in mind in case you have previously used this tool. If you have never used the product administrator, you don't need to worry.*

Additionally, you need to set a name for your Stripe account in your Account settings. You can find more configuration details in the Stripe docs.

Examples

You can find an implementation of these examples on GitHub.

Example 1: One Button

If you're selling a single product, like an eBook for example, you can create a single button that will perform a redirect to the Stripe Checkout page:

Create products and prices

To sell your products, you need to create them in your Stripe account using the Stripe Dashboard or the Stripe API. This is required for Stripe to validate that the request coming from the frontend is legitimate and to charge the correct amount for the selected product/price.

You will need to create both test and live products separately in the Stripe Dashboard. **Make sure you toggle to "Viewing test data", then create your products for local development.**

Create a checkout component that loads Stripe.js and redirects to the checkout

Create a new file at `src/components/checkout.js`. Your `checkout.js` file should look like this. Make sure to add your publishable key in the `loadStripe` method and replace the price ID in the `lineItems` with one of your price IDs from the Stripe dashboard:

```
import React, { useState } from "react"
import { loadStripe } from "@stripe/stripe-js"

const buttonStyles = {
  fontSize: "13px",
  textAlign: "center",
  color: "#000",
  padding: "12px 60px",
  boxShadow: "2px 5px 10px rgba(0,0,0,.1)",
  backgroundColor: "rgb(255, 178, 56)",
  borderRadius: "6px",
  letterSpacing: "1.5px",
}

const buttonDisabledStyles = {
  opacity: "0.5",
  cursor: "not-allowed",
}

let stripePromise
const getStripe = () => {
  if (!stripePromise) {
    stripePromise = loadStripe("<YOUR STRIPE PUBLISHABLE KEY>")
  }
  return stripePromise
}

const Checkout = () => {
  const [loading, setLoading] = useState(false)

  const redirectToCheckout = async event => {
```

```
    event.preventDefault()
    setLoading(true)

    const stripe = await getStripe()
    const { error } = await stripe.redirectToCheckout({
      mode: "payment",
      lineItems: [{ price: "price_1GriHeAKu92npuros981EDUL", quantity: 1 }],
      successUrl: `http://localhost:8000/page-2/`,
      cancelUrl: `http://localhost:8000/`,
    })

    if (error) {
      console.warn("Error:", error)
      setLoading(false)
    }
  }

  return (
    <button
      disabled={loading}
      style={
        loading ? { ...buttonStyles, ...buttonDisabledStyles } : buttonStyles
      }
      onClick={redirectToCheckout}
    >
      BUY MY BOOK
    </button>
  )
}

export default Checkout
```

Note: If you have an older Stripe account with SKU objects instead of prices, you can provide the SKU ID instead:

```
const { error } = await stripe.redirectToCheckout({
  mode: "payment",
  lineItems: [{ price: "sku_DjQJN2HJ1kkvI3", quantity: 1 }],
  successUrl: `http://localhost:8000/page-2/`,
  cancelUrl: `http://localhost:8000/`,
})
```

What did you just do?

You imported React, created a function component that returns a button with some styles, and added a `redirectToCheckout` handler that is executed when the button is clicked. The `getStripe` function returns a Promise that resolves with the Stripe object.

```
let stripePromise
const getStripe = () => {
  if (!stripePromise) {
    stripePromise = loadStripe("< YOUR STRIPE PUBLISHABLE KEY >")
```

```
  }
  return stripePromise
}
```

This identifies you with the Stripe platform, validates the checkout request against your products and security settings, and processes the payment on your Stripe account.

```
const redirectToCheckout = async event => {
  event.preventDefault()
  setLoading(true)

  const stripe = await getStripe()
  const { error } = await stripe.redirectToCheckout({
    mode: "payment",
    lineItems: [{ price: "price_1GriHeAKu92npuros981EDUL", quantity: 1 }],
    successUrl: `http://localhost:8000/page-2/`,
    cancelUrl: `http://localhost:8000/`,
  })

  if (error) {
    console.warn("Error:", error)
    setLoading(false)
  }
}
```

The `redirectToCheckout()` function validates your checkout request and either redirects to the Stripe hosted checkout page or resolves with an error object. Make sure to replace `successUrl` and `cancelUrl` with the appropriate URLs for your application.

```
return (
  <button
    disabled={loading}
    style={
      loading ? { ...buttonStyles, ...buttonDisabledStyles } : buttonStyles
    }
    onClick={redirectToCheckout}
  >
    BUY MY BOOK
  </button>
)
```

Importing the checkout component into the homepage

Now go to your `src/pages/index.js` file. This is your homepage that shows at the root URL. Import your new checkout component in the file underneath the other imports and add your `<Checkout />` component within the `<Layout>` element. Your `index.js` file should now look similar to this:

```
import React from "react"
import { Link } from "gatsby"
```

```
import Layout from "../components/layout"
import Image from "../components/image"
import SEO from "../components/seo"

import Checkout from "../components/checkout" // highlight-line

const IndexPage = () => (
  <Layout>
    <SEO title="Home" keywords={[`gatsby`, `application`, `react`]} />
    <h1>Hi people</h1>
    <p>Welcome to your new Gatsby site.</p>
    <p>Now go build something great.</p>
    <Checkout /> {/* highlight-line */}
    <div style={{ maxWidth: `300px`, marginBottom: `1.45rem` }}>
      <Image />
    </div>
    <Link to="/page-2/">Go to page 2</Link>
  </Layout>
)

export default IndexPage
```

If you go back to `http://localhost:8000/` in your browser and you have `gatsby develop` running, you should now see a big, enticing "BUY MY BOOK" button. C'mon and give it a click!

Example 2: Import products and prices via source plugin

Instead of hardcoding the price IDs, you can use the gatsby-source-stripe plugin to retrieve your prices at build time.

Add the Stripe source plugin

Add the gatsby-source-stripe plugin which you can use to pull in the prices from your Stripe account.

```
npm install gatsby-source-stripe
```

Now you can add the plugin configuration in your `gatsby-config` file:

```
module.exports = {
  siteMetadata: {
    title: `Gatsby E-commerce Starter`,
  },
  plugins: [
    `gatsby-plugin-react-helmet`,
    {
      resolve: `gatsby-source-stripe`,
      options: {
        objects: ["Price"],
        secretKey: process.env.STRIPE_SECRET_KEY,
        downloadFiles: false,
      },
    },
```

```
  ],
}
```

To retrieve your prices from your Stripe account you will need to provide your secret API key. This key needs to be kept secret and must never be shared on the frontend or on GitHub. Therefore you need to set an environment variable to store the secret key. You can read more about the usage of env variables in the [Gatsby docs](#).

In the root directory of your project add a `.env.development` file:

```
# Stripe secret API key
STRIPE_PUBLISHABLE_KEY=pk_test_xxx
STRIPE_SECRET_KEY=sk_test_xxx
```

To use the defined env variable you need to require it in your `gatsby-config.js` or `gatsby-node.js` like this:

```
require("dotenv").config({
  path: `.env.${process.env.NODE_ENV}`,
})
```

Lastly, make sure that your `.gitignore` file excludes all of your `.env.*` files:

```
# dotenv environment variables files
.env
.env.development
.env.production
```

Create a component that lists your products and prices

In your components folder add a new `Products` folder. First, you need a component that queries and lists your prices:

```
import React from "react"
import { graphql, StaticQuery } from "gatsby"

export default function Products(props) {
  return (
    <StaticQuery
      query={graphql`
        query ProductPrices {
          prices: allStripePrice(
            filter: { active: { eq: true } }
            sort: { fields: [unit_amount] }
          ) {
            edges {
              node {
                id
                active
                currency
                unit_amount
```

```
                product {
                  id
                  name
                }
              }
            }
          }
        }
      `}
      render={({ prices }) => (
        <div>
          {prices.edges.map(({ node: price }) => (
            <p key={price.id}>{price.product.name}</p>
          ))}
        </div>
      )}
    />
  )
}
```

You can validate your query and see what data is being returned in GraphiQL, which is available at `http://localhost:8000/___graphql` when running `gatsby develop`.

Once you're happy with your query, create a new page where you can import the newly created `Products` component:

```
import React from "react"

import Layout from "../components/layout"
import SEO from "../components/seo"

import Products from "../components/Products/Products" // highlight-line

const AdvancedExamplePage = () => (
  <Layout>
    <SEO title="Advanced Example" />
    <h1>This is the advanced example</h1>
    <Products /> {/* highlight-line */}
  </Layout>
)

export default AdvancedExamplePage
```

When navigating to `http://localhost:8000/advanced/` you should now see a list of paragraphs with your product names.

Extract loading of Stripe.js into a utility function

When using Stripe.js across multiple pages and components it is recommended to extract `loadStripe` into a utility function that exports a `getStripe` singleton:

```
/**
 * This is a singleton to ensure we only instantiate Stripe once.
 */
import { loadStripe } from "@stripe/stripe-js"

let stripePromise
const getStripe = () => {
  if (!stripePromise) {
    stripePromise = loadStripe(process.env.GATSBY_STRIPE_PUBLISHABLE_KEY)
  }
  return stripePromise
}

export default getStripe
```

Create a component that represents a single product

To make your products more visually appealing and interactive, create a new `ProductCard` component in your `Products` folder:

```
import React, { useState } from "react"
import getStripe from "../../utils/stripejs"

const cardStyles = {
  display: "flex",
  flexDirection: "column",
  justifyContent: "space-around",
  alignItems: "flex-start",
  padding: "1rem",
  marginBottom: "1rem",
  boxShadow: "5px 5px 25px 0 rgba(46,61,73,.2)",
  backgroundColor: "#fff",
  borderRadius: "6px",
  maxWidth: "300px",
}
const buttonStyles = {
  display: "block",
  fontSize: "13px",
  textAlign: "center",
  color: "#000",
  padding: "12px",
  boxShadow: "2px 5px 10px rgba(0,0,0,.1)",
  backgroundColor: "rgb(255, 178, 56)",
  borderRadius: "6px",
  letterSpacing: "1.5px",
}

const buttonDisabledStyles = {
  opacity: "0.5",
  cursor: "not-allowed",
}
```

```
const formatPrice = (amount, currency) => {
  let price = (amount / 100).toFixed(2)
  let numberFormat = new Intl.NumberFormat(["en-US"], {
    style: "currency",
    currency: currency,
    currencyDisplay: "symbol",
  })
  return numberFormat.format(price)
}

const ProductCard = ({ product }) => {
  const [loading, setLoading] = useState(false)

  const handleSubmit = async event => {
    event.preventDefault()
    setLoading(true)

    const price = new FormData(event.target).get("priceSelect")
    const stripe = await getStripe()
    const { error } = await stripe.redirectToCheckout({
      mode: "payment",
      lineItems: [{ price, quantity: 1 }],
      successUrl: `${window.location.origin}/page-2/`,
      cancelUrl: `${window.location.origin}/advanced`,
    })

    if (error) {
      console.warn("Error:", error)
      setLoading(false)
    }
  }

  return (
    <div style={cardStyles}>
      <form onSubmit={handleSubmit}>
        <fieldset style={{ border: "none" }}>
          <legend>
            <h4>{product.name}</h4>
          </legend>
          <label>
            Price{" "}
            <select name="priceSelect">
              {product.prices.map(price => (
                <option key={price.id} value={price.id}>
                  {formatPrice(price.unit_amount, price.currency)}
                </option>
              ))}
            </select>
          </label>
        </fieldset>
        <button
```

```
          disabled={loading}
          style={
            loading
              ? { ...buttonStyles, ...buttonDisabledStyles }
              : buttonStyles
          }
        >
          BUY ME
        </button>
      </form>
    </div>
  )
}

export default ProductCard
```

This component renders a neat card for each individual product, a dropdown to select the specific price for the product, nicely formatted pricing, and a "BUY ME" button. The button triggers the `handleSubmit` which gets the price ID from the dropdown select and then redirects to Stripe Checkout.

Lastly, you need to refactor your `Products` component to group the prices by their products and create a `ProductCard` for each product:

```
import React from "react"
import { graphql, StaticQuery } from "gatsby"
import ProductCard from "./ProductCard" //highlight-line

const containerStyles = {
  display: "flex",
  flexDirection: "row",
  flexWrap: "wrap",
  justifyContent: "space-between",
  padding: "1rem 0 1rem 0",
}

const Products = () => {
  return (
    <StaticQuery
      query={graphql`
        query ProductPrices {
          prices: allStripePrice(
            filter: { active: { eq: true } }
            sort: { fields: [unit_amount] }
          ) {
            edges {
              node {
                id
                active
                currency
                unit_amount
                product {
```

```
                id
                name
              }
            }
          }
        }
      }
    `}
    render={({ prices }) => {
      // highlight-start
      // Group prices by product
      const products = {}
      for (const { node: price } of prices.edges) {
        const product = price.product
        if (!products[product.id]) {
          products[product.id] = product
          products[product.id].prices = []
        }
        products[product.id].prices.push(price)
      }

      return (
        <div style={containerStyles}>
          {Object.keys(products).map(key => (
            <ProductCard key={products[key].id} product={products[key]} />
          ))}
        </div>
      )
      // highlight-end
    }}
  />
)
}

export default Products
```

Adding shopping cart functionality

To add shopping cart functionality to your Gatsby site, you can use the use-shopping-cart library. It allows you to keep the cart state across components and pages, and even stores the cart state in `localStorage`. You can find a Gatsby example in their GitHub repository.

Testing Payments

In test mode (when using the API key that includes *test*) Stripe provides test cards for you to test different checkout scenarios.