

Esta clase va a ser

- grabada
a

Clase 46. PROGRAMACIÓN BACKEND

Pasarelas de pago

Temario

45

Product Cloud: Despliegue de nuestro aplicativo

- ✓ Despliegue de nuestro aplicativo
- ✓ Configuración de pipeline en Railway.app

46

Pasarelas de pago

- ✓ Cerramos nuestra app

Objetivos de la clase

- Analizar las características de una pasarela de pago para ejecutar cobros.
- Armar el backend para procesar una pasarela de pago conectada con el frontend.

CLASE N°45

Glosario

Railway.app: es una de las alternativas que podemos utilizar para realizar el despliegue de nuestra aplicación.

Pipeline: todo el flujo que comprende un proceso. Una característica importante de un pipeline es el uso de diferentes **stages**.

MAPA DE CONCEPTOS



Cerramos nuestra aplicación:
pasarela de pago

¡Felicidades!

Si estás leyendo esto, estás en la última clase de tu curso, lo cual significa que ya tienes un servidor que cuenta con las siguientes características:

- ✓ Un sistema de ruteo con controladores.
- ✓ Una arquitectura basada en capas aplicando patrones DAO y Repository para poder consumir correctamente tus servicios.
- ✓ Manejo de sesiones y sistema de mailing.
- ✓ Pruebas de estrés, testing unitario y de integración.
- ✓ Despliegue de tu aplicativo.

¡Mira cuán lejos has llegado!



Ahora, cerramos con broche de oro

Durante este tiempo estuvimos construyendo un ecommerce, sin embargo, ¿no sientes que nos falta algo?

¡El cobro es la parte más importante de un ecommerce!

Nuestro botón de “finalizar compra” se siente algo vacío, si no sentimos que fluye dinero de por medio.

El día de hoy nos centraremos en una pasarela de pago.

Existen diferentes pasarelas de pago, por ejemplo Stripe, usado por empresas internacionales, y Mercado Pago, muy popular en Argentina.



¿Cómo funcionan los pagos por internet?

Cuando un cliente visita nuestro ecommerce, debe tener plena seguridad de que el procesamiento se hará con la mayor discreción y control que sea posible.

Una pasarela de pago será un intermediario entre nuestro cliente y nosotros, el cual cuenta con un sistema propio de validación, procesamiento, post-procesamiento y gestión general de los pagos que hace el cliente, desde el momento en el que éste ingresa un método de pago, hasta el momento en el que el pago resulte “Exitoso” y se haga el cargo oficialmente a la cuenta bancaria asociada al cliente.



Funcionamiento interno de una pasarela de pago

La pasarela de pago necesita tener un cliente al cual asociar los movimientos (En este caso el cliente seremos nosotros, recuerda que el servicio es externo y nosotros sólo lo estamos solicitando).

Una vez que se cuenta con dicho cliente, bajo la configuración del cliente éste podrá configurar qué métodos de pago son aceptados en su sitio web. (El método de pago por defecto es una tarjeta de crédito/débito).

El usuario que utilice nuestro sitio web podrá seleccionar alguno de esos métodos habilitados para su cuenta e intentará procesar el pago.

La pasarela de pago evaluará la validez de dicho método de pago, en múltiples aspectos según sea el método, para la tarjeta los más comunes son:

- ✓ Número de tarjeta válido.
- ✓ CVC/CVV de tarjeta válido.
- ✓ Expiración de tarjeta válida.
- ✓ Fondos de tarjeta válidos.

¡Si todo está bien...!

Entonces la pasarela de pago podrá aprobarlo y relacionar dicho movimiento a la cuenta que asociamos a ésta (es decir, el dinero ya se podrá ver reflejado en la cuenta configurada internamente por nosotros).

Además, éste devolverá información importante para que nosotros podamos mostrar algún mensaje de **"su pago fue realizado correctamente"**, brindando la tranquilidad al usuario de que su pago fue exitoso y puede proceder a recibir la información de su producto.

No tenemos que hacer algún movimiento adicional con los montos indicados (al menos no en los pagos básicos y cerrados), de manera que con la información que nos proporcione la pasarela de pago, podremos llenar nuestra base de datos o actualizar lo que consideremos pertinente.

La pasarela hace el trabajo sucio, y nosotros sólo mostramos el resultado final.

¿Y si algo sale mal?

Al procesar un pago, un error se puede dar de tres maneras:

- ✓ Error desde el backend: La pasarela procesó correctamente el pago, pero debido a algún fallo de control nuestro servidor arroja un error. En este caso, la pasarela ya procesó el pago y tendremos que buscar la forma de relacionarlo sin afectar al cliente (porque, en realidad éste ya pagó).
- ✓ Error desde el frontend: La pasarela procesó correctamente el pago y el backend también, sin embargo, el front no muestra info del pago finalizado y ocasiona que el cliente no sepa si se puede proceder o no. Ésto puede llevar a malos entendidos que, en muchos casos pueden terminar en reembolsos y una mala reputación por parte de nuestra página.
- ✓ Error en la pasarela de pago: Es un error bastante inusual, sin embargo, cuando la pasarela llega a tener algún error, normalmente se da al momento de procesar el pago, de manera que habitualmente no hay que arreglar nada con el cliente, ya que no se le afectó, sólo hay que informarle que el servicio está inactivo temporalmente.

Importante

Así como el manejo de sesiones de un usuario, la gestión de pagos es uno de los módulos más serios que se deben tocar en un sistema, ¡mucho cuidado al utilizarlo! **Recuerda que estás poniendo en juego la integridad del cliente más allá de la información, sino en cuestión monetaria.**

¡Ocupa tu máxima concentración en este proceso módulos!

**Antes de configurar
nuestro primer pago**

Un pago ocupa un frontend

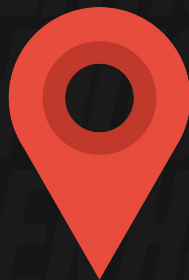
Para poder desarrollar la clase, necesitaremos un proyecto que nos permita contar con un frontend desarrollado y configurado para trabajar con las plataformas de pago que utilizaremos.

Para ello, te brindaremos en [este enlace](#) un proyecto con las dos pasarelas que utilizaremos en clase.

El proyecto está desarrollado en React js y contiene sólo lo necesario para el procesamiento de un pago.

Vamos a clonar el proyecto y lo echaremos a andar con

npm install + npm start



Checkpoint: Espacio de preparación

Toma un tiempo para clonar el proyecto [aquí](#). Revisa el código e instala las dependencias indicadas.

Tiempo estimado: **5 minutos**

Entendiendo la estructura de Stripe

¿Qué es Stripe?

Stripe es una pasarela de pago pensada para procesar los movimientos bajo la lógica que recientemente mencionamos.

Es uno de los procesadores de pago **de nivel internacional** más utilizado, de manera que muchas empresas la utilizan como componente principal para procesar dichos movimientos.

Debido a su carácter internacional, será la plataforma utilizada para realizar el flujo de nuestros pagos en esta clase.



Terminología elemental de Stripe

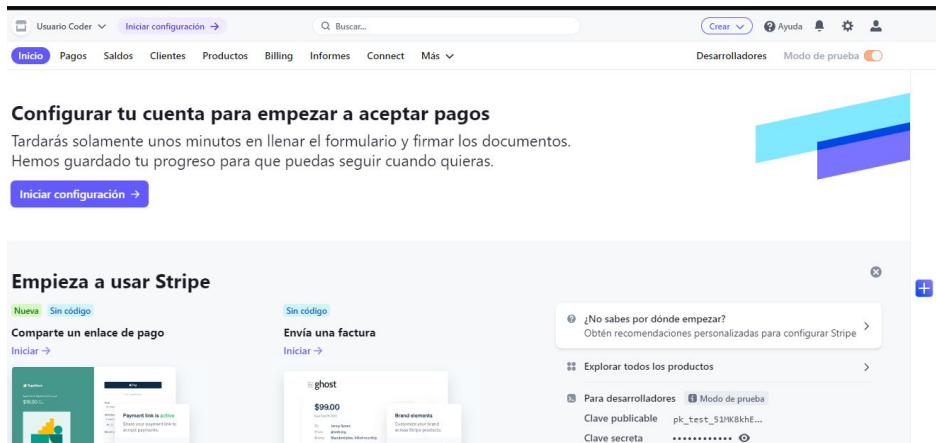
- ✓ Método de pago: Hace referencia a todos los métodos que el cliente puede utilizar para pagar, el método por defecto siempre es la tarjeta de crédito/débito.
- ✓ Intento de pago (Payment Intent): Registro único que representa cada intento que el usuario está utilizando para finalizar su proceso de compra.
- ✓ `clientSecret`: Código único de cliente relacionado con un intento de pago.
- ✓ Método de pago (Payment Method): Hace referencia a la alternativa que se está utilizando para procesar el pago.
- ✓ Elements: Componentes de Stripe pensados para poder representar los diferentes procesos de pago (`CardElement`, `PaymentElement`, etc)

Lo primero, registrarse

Lo primero será crear una cuenta de Stripe para poder recibir los movimientos de prueba que haremos.

Nota que Stripe no está activo en todos los países. **Esto no es una limitante real**, ya que podemos registrarnos con cuenta Estadounidense, por ejemplo. (Recuerda que las empresas utilizan en su mayoría métodos internacionales, por lo que no es bueno enfrascarse en un método nacional únicamente).

El objetivo es llegar a nuestro panel principal, en la pantalla de inicio, para comenzar con nuestras respectivas configuraciones.



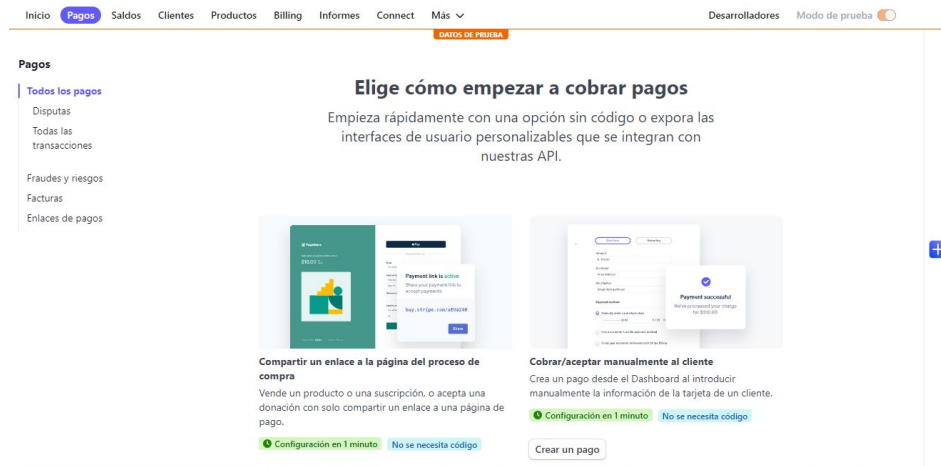
Panel de pagos

Aquí podremos visualizar todos los movimientos que sean realizados en nuestra página.

Podemos ver y analizar:

- ✓ Los montos que se han pagado
- ✓ El estatus de dichos pagos
- ✓ La información adicional referente a algún pago.
- ✓ La posibilidad de realizar reembolsos ante alguna problemática sobre el movimiento de algún cliente.

Éste y tu panel de desarrollador son los elementos más importantes para que puedas llevar un seguimiento completo de los movimientos de tu sitio web.



Buscamos la opción “desarrolladores”

En la sección de desarrolladores necesitaremos dos claves:

La clave publicable hace referencia a la clave que aparecerá en nuestro respectivo **frontend**.

La clave secreta está ligada directamente al **backend**, con el fin de llevar a cabo el procesamiento trasero que implique un pago según sea la empresa.



Inicio Pagos Saldos Clientes Productos Billing Informes Connect Más ▾ Desarrolladores

DATOS DE PRUEBA

Desarrolladores

- Resumen
- Claves de API**
- Webhooks
- Eventos
- Registros
- Aplicaciones

Claves de API

Estás viendo las claves de API de prueba. Alternar para ver las claves activas. ☐ Ver claves activas

Claves estándar

Estas claves te permitirán autenticar las peticiones de API. [Más información](#)

NOMBRE	TOKEN	ÚLTIMA VEZ QUE SE USÓ	DE CREACIÓN
Clave publicable	pk_test_51fK8khEa2mQfWfWGDQkoy6oY2iBguE'lgzd751XPW7TZgXx5UNQaNgT13P1ZlHq0nzZFvtNKH7KVZCDGcd8Ap617p004x0ZZzYV	—	28 dic.
Clave secreta	<input type="button" value="Revelar la clave de prueba"/>	—	28 dic.

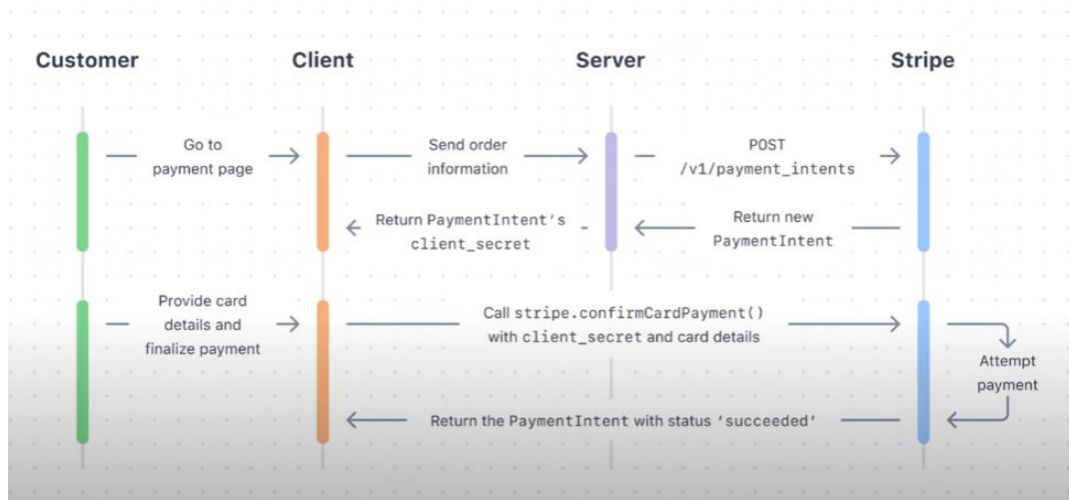
Aprendiendo sobre un flujo de pago

El flujo recomendado para el proceso de un pago en Stripe es el siguiente:

- ✓ Una capa Customer que representa **la acción real del cliente**.
- ✓ Una capa Client que representa el frontend.
- ✓ Una capa server que representa al backend.
- ✓ Una capa Stripe que representa la pasarela de pago de manera interna.

Ahondaremos en este proceso a partir de las capturas del frontend que tenemos, **mientras armamos el backend nosotros desde cero**
(Te recomendamos ir levantando una aplicación básica de express)

Aprendiendo sobre un flujo de pago



Seguimiento de un flujo de pago completo



Customer: Entrar a comprar.

Para la simulación de este proceso de compra, se tiene un panel con 5 productos a seleccionar. El producto seleccionado será el producto a comprar.

Se utiliza un arreglo con datos de los productos y sólo se comprará **un producto a la vez**, esto para no acomplejar más las cosas de lado del front (concentrémonos en el backend).

Stripe

papas
1000

queso
500

hamburguesa
1500

soda
1000

golosinas
800

```
const mockCart = [  
  { id: 1, name: "papas", price: 1000 },  
  { id: 2, name: "queso", price: 500 },  
  { id: 3, name: "hamburguesa", price: 1500 },  
  { id: 4, name: "soda", price: 1000 },  
  { id: 5, name: "golosinas", price: 800 }  
]
```



Client: Inicialización de Stripe

Desde que el cliente ha entrado al panel para procesar la compra, nosotros iremos desarrollando la conexión del frontend con Stripe, para esto, utilizaremos una **stripePromise**, la cual cargará todos los elementos necesarios para que Stripe funcione con el front

Para que esta parte funcione, recuerda cambiar el valor de la variable de entorno en el archivo .env del frontend

```
11  const stripePromise = loadStripe(process.env.REACT_APP_STRIPE_KEY);  
12
```

recursopasarelapago > .env

```
1  REACT_APP_BASE_URL = http://localhost:8080  
2  REACT_APP_PAYMENT_ENDPOINT = /api/payments  
3  REACT_APP_STRIPE_KEY = AQUI TU PUBLIC KEY DE STRIPE
```



Client: producto seleccionado

```
const getClientSecret = async () => {  
  console.log(currentProduct);  
  const service = new PaymentService();  
  service.createPaymentIntent({product:  
}  
currentProduct&&getClientSecret();
```

Archivo de frontend en: **src/pages/Stripe/Stripe.jsx**

```
createPaymentIntent = ({productId, callbackSuccess, callbackError}) => {  
  const requestInfo = {url: `${REACT_APP_BASE_URL}${REACT_APP_PAYMENT_ENDPOINT}/payment-intents?id=${productId}`};  
  this.client.makePostRequest(requestInfo);  
}
```

Archivo de frontend en: **src/services/paymentService.js**

¿Cómo reacciona el frontend ante la selección de un producto? (recuerda que para ti será un carrito de compra completo).

En cuanto el frontend detecta que se ha seleccionado un producto, mandará a llamar el servicio **getClientSecret**, el cual conectará con nuestro backend, enviando la información del producto que queremos desarrollar.

Para esto, se hará una petición POST, donde se enviará por query params el id del producto, para generar el cobro.



Ejemplo en vivo

¡Es nuestro turno!

El profesor desarrollará el endpoint `payment-intents` en el router `/api/payments` con el fin de crear el pago en Stripe

Duración: **20 minutos**

Creamos una aplicación de backend e instalamos las dependencias.

Se levanta un aplicativo basado en express y se crea una estructura de router (No es necesario crear controller), las dependencias a utilizar son:

```
"dependencies": {  
  "cors": "^2.8.5",  
  "express": "^4.18.2",  
  "stripe": "^11.5.0"  
}
```

Al final, no necesitaremos mayor estructura que la que tenemos en la captura que se muestra a continuación.

Ahora nos concentraremos en el desarrollo del endpoint **payment-intents**

```
JS app.js  X  
stripebackend > src > JS app.js > ...  
1  import express from 'express';  
2  import cors from 'cors';  
3  import paymentRouter from './routes/payments.router.js';  
4  
5  const app = express();  
6  app.use(cors());  
7  app.use('/api/payments', paymentRouter);  
8  app.listen(8080, () => console.log(`Listening on 8080`));
```

 EJEMPLO: /api/payments/payment-intents

payment-intents endpoint: Parte I

Con el fin de no trabajar una base de datos, tomaremos el mismo arreglo de prueba que tiene el frontend, esto para que coincida el id enviado desde el servicio.

Nota que creamos un `paymentIntentInfo`, donde colocamos el monto del producto que deseamos crear, `currency` es el tipo de cambio que estamos utilizando en la plataforma.

```
JS payments.router.js X
stripebackend > src > routes > JS payments.router.js > router.post('/payment-intents') callback > paymentIntentInfo > currency

1 import {Router} from 'express';
2 import PaymentService from '../services/payments.js'
3
4 const router = Router();
5
6 //Los productos coinciden con las pruebas del frontend.
7 const products = [
8   { id: 1, name: "papas", price: 1000 },
9   { id: 2, name: "queso", price: 500 },
10  { id: 3, name: "hamburguesa", price: 1500 },
11  { id: 4, name: "soda", price: 1000 },
12  { id: 5, name: "golosinas", price: 800 }
13 ]
14
15 router.post('/payment-intents', async (req, res) => {
16   const productRequested = products.find(product => product.id === parseInt(req.query.id))
17   if (!productRequested) return res.status(404).send({ status: "error", error: "product not found" });
18   const paymentIntentInfo = {
19     amount: productRequested.price,
20     currency: 'usd'
21   }
22   const service = new PaymentService();
23   let result = await service.createPaymentIntent(paymentIntentInfo);
24   console.log(result);
25   res.send({status:"success",payload:result})
26 })
27 export default router;
```


 EJEMPLO: /api/payments/payment-intents

payment-intents endpoint: Parte II

Ahora, llamamos a un PaymentService para que genere el paymentIntent a partir del método

service.createPaymentIntent,

Esto significa que crearemos un archivo en

services/payments.js para desarrollar el método utilizado.

```
JS payments.router.js X
stripebackend > src > routes > JS payments.router.js > router.post('/payment-intents') callback > paymentIntentInfo > currency

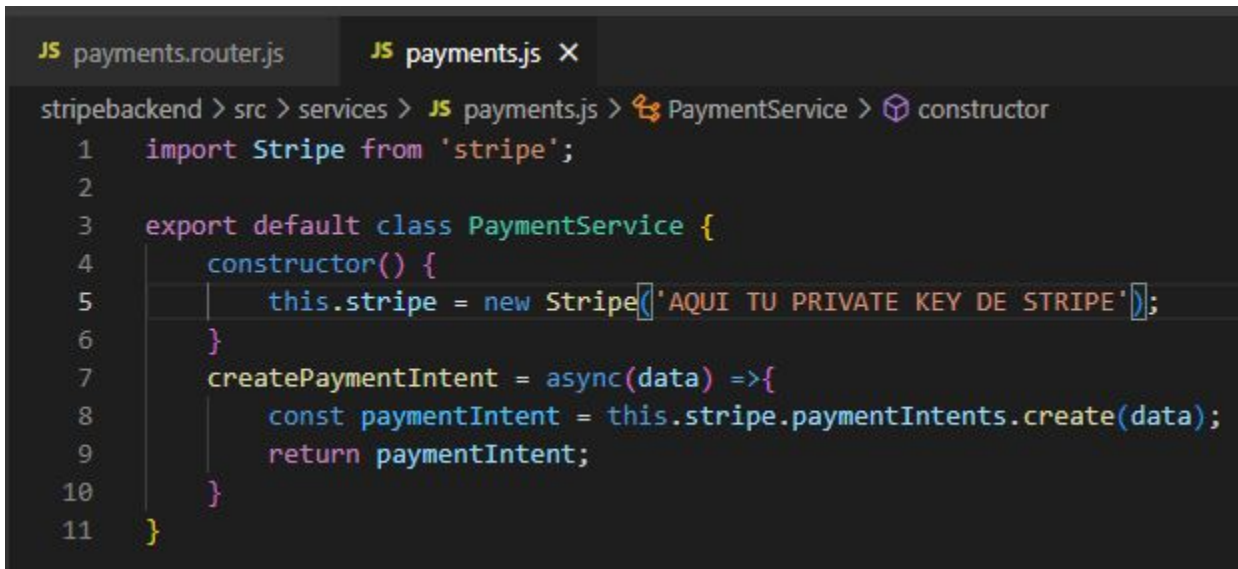
1 import {Router} from 'express';
2 import PaymentService from '../services/payments.js'
3
4 const router = Router();
5
6 //Los productos coinciden con las pruebas del frontend.
7 const products = [
8   { id: 1, name: "papas", price: 1000 },
9   { id: 2, name: "queso", price: 500 },
10  { id: 3, name: "hamburguesa", price: 1500 },
11  { id: 4, name: "soda", price: 1000 },
12  { id: 5, name: "golosinas", price: 800 }
13 ]
14
15 router.post('/payment-intents', async (req, res) => {
16   const productRequested = products.find(product => product.id === parseInt(req.query.id))
17   if (!productRequested) return res.status(404).send({ status: "error", error: "product not found" });
18   const paymentIntentInfo = {
19     amount: productRequested.price,
20     currency: 'usd'
21   }
22   const service = new PaymentService();
23   let result = await service.createPaymentIntent(paymentIntentInfo);
24   console.log(result);
25   res.send({status:"success",payload:result})
26 })
27 export default router;
```

PaymentService

Crearemos una carpeta **services** en src y desarrollamos la siguiente clase para el servicio.

Nota que al inicializarse, también se inicializa stripe con la **private key** que tenemos en la plataforma.

Nota cómo realmente stripe es quien lo hace todo de manera interna, sólo estamos solicitando que se cree un intento de pago por parte de la plataforma, y el resultado lo enviaremos al frontend.



```
JS payments.router.js    JS payments.js X
stripebackend > src > services > JS payments.js > PaymentService > constructor
1  import Stripe from 'stripe';
2
3  export default class PaymentService {
4    constructor() {
5      this.stripe = new Stripe('AQUI TU PRIVATE KEY DE STRIPE');
6    }
7    createPaymentIntent = async(data) =>{
8      const paymentIntent = this.stripe.paymentIntents.create(data);
9      return paymentIntent;
10   }
11 }
```




EJEMPLO: Envío de información

El frontend necesita un formato específico.

El formato en el que se lee la información en el front es: `res.data.payload.client_secret`

Ergo, sabemos que la parte de **res.data** viene por parte de axios, mientras que el **payload.client_secret** depende de nosotros, para que el frontend pueda tomar el `client_secret`, necesitamos enviar el **paymentIntent** exactamente como lo generó stripe.

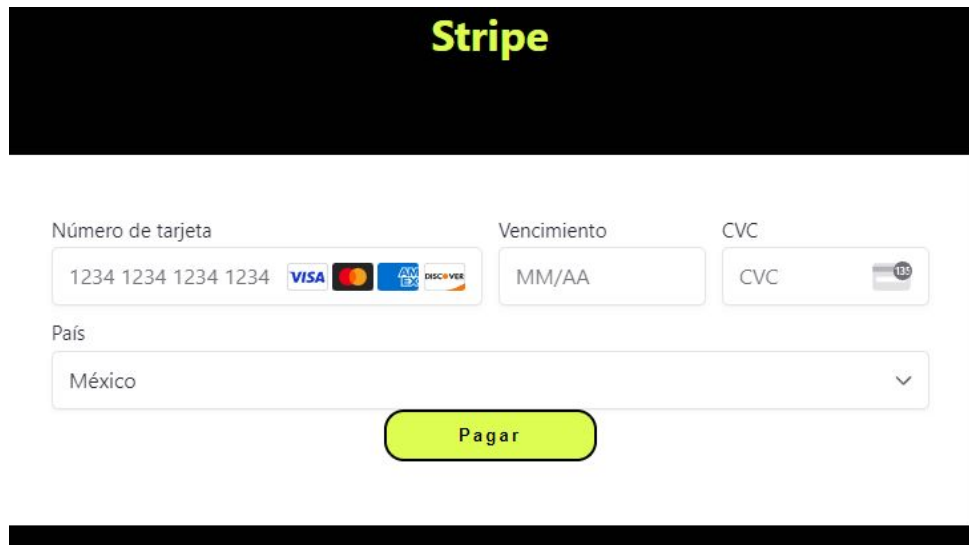
Es muy importante que no se modifique la respuesta al enviar al frontend.

 EJEMPLO: Probando la selección de producto.

Seleccionando el producto nuevamente.

Una vez que el frontend tiene un endpoint funcional al cual enviar la información. Entramos al panel de selección de productos inicial y elegimos cualquier producto de nuestra elección. Una vez seleccionado, se mandará a llamar al servicio **payment-intents** de nuestro backend, y una vez teniendo un `paymentIntent` otorgado por nuestro backend, notaremos cómo **el panel avanza al panel de pago.**

Una vez llegando a este punto, podemos decir que nuestro backend está finalizado.



The image shows a Stripe payment form interface. At the top, the word "Stripe" is displayed in a green font on a black background. Below this, the form is divided into several sections. The first section contains three input fields: "Número de tarjeta" (Card Number) with the placeholder "1234 1234 1234 1234" and logos for VISA, Mastercard, AMEX, and Discover; "Vencimiento" (Expiration) with the placeholder "MM/AA"; and "CVC" with the placeholder "CVC" and a small card icon. Below these is a "País" (Country) dropdown menu showing "México" with a downward arrow. At the bottom of the form is a large green button labeled "Pagar" (Pay).



Break

¡10 minutos y volvemos!

Analizando el cierre de pago

¿Qué ocurre al crear un paymentIntent?

Cuando seleccionamos el producto y el backend hizo el paymentIntent, Stripe de manera interna lo registró en la plataforma. Esto quiere decir que se comenzó a procesar un pago. El estatus **Incompleto**, indica que no se ha recibido la información del pago aún (El pago no se ha finalizado o se cerró antes de procesar).

DATOS DE PRUEBA

Pagos

Todos los pagos

Disputas

Todas las transacciones

Fraudes y riesgos

Facturas

Enlaces de pagos

Pagos

Todos

Exitosos

Rembolsados

No capturados

Error

+ Fecha + Importe + Estatus + Método de pago

× Borrar los filtros

<input type="checkbox"/>	IMPORTE	DESCRIPCIÓN	CLIENTE	FECHA
<input type="checkbox"/>	5,00 US\$	Incompleto 	pi_3MKkVKEa2mQfMMGD00Z1DCPX	30 dic. 9:28 ...

1 resultado

Anterior

Siguiente

Exportar

+ Crear pago

¿Qué ocurre al crear un paymentIntent?

Si refrescamos la página y seleccionamos otro producto, notaremos que nos creará un payment intent completamente diferente, pues por seguridad **nunca** relaciona un payment intent previo.

Ahora, desde el frontend completamos el pago de los 15 USD

Pagos

Todos

Exitosos

Rembolsados

No capturados

Error

+ Fecha

+ Importe

+ Estatus

+ Método de pago



IMPORTE

DESCRIPCIÓN



15,00 US\$

Incompleto



pi_3MKkjpEa2mQfMMGD0ePsH8iM



5,00 US\$

Incompleto



pi_3MKkVKEa2mQfMMGD00Z1DCPX

¿Cómo probar un pago?

Ya que utilizar métodos de pago reales puede afectar la integridad de nuestra economía (porque nadie quiere perder 15 USD sólo para probar código), Stripe nos brinda un listado completo de elementos para probar diferentes casos, algunos de los elementos que se pueden probar son:

- ✓ Pagos con tarjeta exitosos.
- ✓ Pagos con tarjeta fallidos:
 - Por Código de seguridad incorrecto.
 - Por tarjeta expirada.
 - Por número de tarjeta incorrecto.
 - Por Fondos insuficientes.
- ✓ Pagos con transferencias bancarias.

Puedes revisar todas las posibilidades de testing en [este link](#)

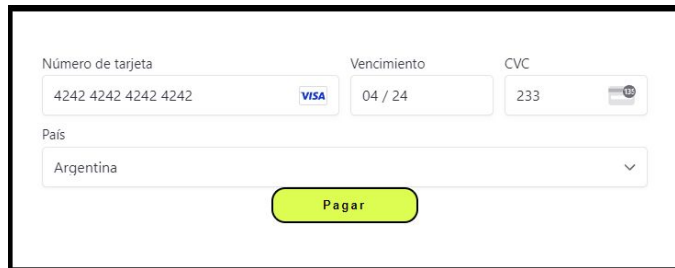
Números más utilizados

A continuación se te dejan algunos de los números de tarjeta más utilizados para probar rechazos:

- ✓ Pago válido: 4242424242424242
- ✓ Rechazo por fondos insuficientes: 40000000000009995
- ✓ Rechazo por tarjeta expirada: 4000000000000069
- ✓ Rechazo por código de seguridad incorrecto: 4000000000000127
- ✓ Error de procesamiento (equivalente a error interno): 4000000000000119

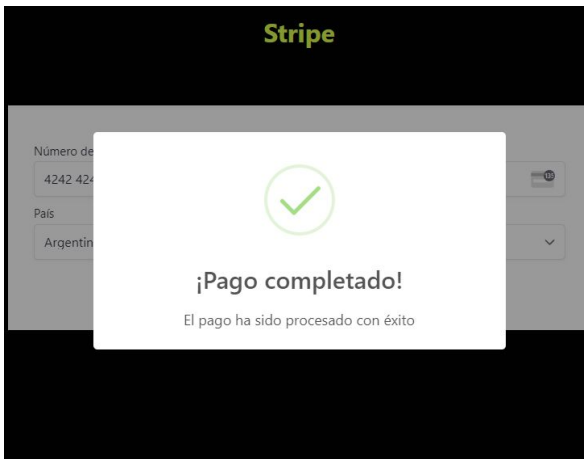
Intentaremos procesar un pago "correcto"

Notamos que, al colocar datos de prueba correctos, el pago finaliza correctamente y la plataforma de Stripe lo tomará como "Pago exitoso". ¡Hemos hecho un pago efectivo!



Formulario de pago de Stripe. Campos:

- Número de tarjeta: 4242 4242 4242 4242 (Visa)
- Vencimiento: 04 / 24
- CVC: 233
- País: Argentina
- Botón: Pagar



Pagos

Todos	Exitosos	Rebolsados	No capturados	Error
➕ Fecha	➕ Importe	➕ Estatus	➕ Método de pago	
<input type="checkbox"/> IMPORTE	DESCRIPCIÓN			
<input type="checkbox"/> 15,00 US\$	Exitoso ✓	pi_3MKkjpEa2mQfMwMGD0ePsH8iM		
<input type="checkbox"/> 5,00 US\$	Incompleto ⚠	pi_3MKkVKEa2mQfMwMGD00ZlDCPX		

Importante

Realiza todas las pruebas que te sea posible, debido a que una vez que entramos en un entorno productivo, Stripe bloquea todos los números previamente proporcionados.

¡Haz que el proceso sea lo más seguro posible!

Uso de metadata en un pago

La metadata es información muy útil para poder guardar información más allá de la proporcionada por Stripe. Esto permite que en el pago se guarden otras cosas **ajenas a stripe, pero cruciales para nosotros**.

¿Qué tal si deseo guardar la referencia de algún usuario, de algún negocio, los detalles de la compra?

Este tipo de información puede ser guardada en la metadata

Agregaremos al servicio de Stripe información sobre el pago **hardcodeada**, con el fin de ver cómo se refleja esto en la plataforma.

Modificando payment Intent

```
router.post('/payment-intents', async (req, res) => {  
  const productRequested = products.find(product => product.id === parseInt(req.query.id))  
  if (!productRequested) return res.status(404).send({ status: "error", error: "product not found" });  
  const paymentIntentInfo = {  
    amount: productRequested.price,  
    currency: 'usd',  
    metadata: {  
      userId: "Id autogenerado por Mongo",  
      orderDetails: JSON.stringify({  
        //Suponiendo que en los detalles viene la cantidad a comprar del producto (que tú ya tendrás en tu carrito)  
        [productRequested.name]: 2  
      }, null, '\t'),  
      //Dirección para tener referencia de dónde se envió el producto final.  
      address: JSON.stringify({  
        street: "calle de prueba",  
        postalCode: "39441",  
        externalNumber: "123123"  
      }, null, '\t')  
    }  
  }  
})
```

Realizamos un pago

Una vez hechos los cambios, vamos a generar un pago para ver qué es lo que se guarda dentro de Stripe.

Al abrir el pago, podremos visualizar cómo en la información del pago, aparecen los metadatos guardados

¡Y listo! Ahora cualquier pago, ante cualquier fallo o aclaración, tenemos información adicional sobre el pago para poder corroborar.

Pago iniciado
30 dic. 2022 10:21

DATOS DE PRUEBA

Datos del pago

Importe	8.00 US\$
Estado	Incompleto
Descripción	No hay descripción. ✎ Modificar

Metadatos

[✎ Editar](#)

address	{ "street": "calle de prueba", "postalCode": "39441", "externalNumber": "123123" }
orderDetails	{ "golosinas": 2 }
userId	Id autogenerado por Mongo



Backend de una aplicación ecommerce



Backend de una aplicación ecommerce

Desde el router de `/api/users`, crear tres rutas:

- ✓ GET / deberá obtener todos los usuarios, éste sólo debe devolver los datos principales como nombre, correo, tipo de cuenta (rol)
- ✓ DELETE / deberá limpiar a todos los usuarios que no hayan tenido conexión en los últimos 2 días. (puedes hacer pruebas con los últimos 30 minutos, por ejemplo). Deberá enviarse un correo indicando al usuario que su cuenta ha sido eliminada por inactividad

Crear una vista para poder visualizar, modificar el rol y eliminar un usuario. Esta vista únicamente será accesible para el administrador del ecommerce



Backend de una aplicación ecommerce

Modificar el endpoint que elimina productos, para que, en caso de que el producto pertenezca a un usuario premium, le envíe un correo indicándole que el producto fue eliminado.

Finalizar las vistas pendientes para la realización de flujo completo de compra. NO ES NECESARIO tener una estructura específica de vistas, sólo las que tú consideres necesarias para poder llevar a cabo el proceso de compra.

No es necesario desarrollar vistas para módulos que no influyan en el proceso de compra (Como vistas de usuarios premium para crear productos, o vistas de panel de admin para updates de productos, etc)

Realizar el despliegue de tu aplicativo en la plataforma de tu elección (Preferentemente Railway.app, pues es la abarcada en el curso) y corroborar que se puede llevar a cabo un proceso de compra completo.



Última entrega

Objetivos generales

- ✓ Completar el proyecto final

Objetivos específicos

- ✓ Conseguir una experiencia de compra completa
- ✓ Cerrar detalles administrativos con los roles.

Formato

- ✓ Link al repositorio sin node_modules
- ✓ Link del proyecto desplegado..

Sugerencias

- ✓ Presta especial atención a las rúbricas de Proyecto final. ¡Es crucial para alcanzar la nota que esperas!
- ✓ Debido a la complejidad de frontend requerida para poder aplicar una pasarela de pago, el PF no evalúa la pasarela de pago.

¿Preguntas?

Muchas gracias.

Resumen de la clase hoy

- ✓ Despliegue de aplicación
- ✓ Despliegue con Railway.app
- ✓ Pipeline de despliegue

Opina y valora
esta clase

#DemocratizandoLaEducación