

3. INTEROPERACIÓN: API de Realizar Pago

3.1 INTRODUCCIÓN

Nuestro proyecto expone la funcionalidad de permitir a proyectos externos el realizar pagos a través de nuestra pasarela de pago. Para ello, se exponen dos endpoints principales:

1. **“/pago/form”**: endpoint encargado de mostrar los detalles de la compra, así como mostrar el formulario para completar el pago.

The screenshot displays a payment interface for 'TPVV BoarDalo'. It is divided into two main sections: 'Datos de la Compra' (Purchase Data) and 'Pago con tarjeta' (Card Payment).

Datos de la Compra:

Importe:	888,89 €
Comercio:	Tienda Online v2
Nº de pedido:	TICKET-888
Fecha:	09/09/2029
Hora:	13:45

Pago con tarjeta:

Nombre Completo:

Nº Tarjeta:

Caducidad:

Cód. Seguridad:

Figura 1: Datos de la pasarela de Pago

2. **“/pago/realizar”**: endpoint utilizado para registrar un pago en la base de datos del TPVV a partir de los datos introducidos. Este método se encarga de manejar la recepción de la API Key pasada por cabecera HTTP para obtener el comercio asociado a la hora de registrar el pago

3.2 PROYECTO BASE PARA LA INTERCONEXIÓN

Una de nuestras preocupaciones iniciales era el cómo poder probar el correcto funcionamiento de la funcionalidad de “**realizar pago**” sin tener que esperar a que estuviese completada la implementación del resto de grupos. En base a esta motivación, tomamos la decisión de construir un proyecto base independiente del TPVV. Dicho proyecto simularía una **tienda online**, en la que se muestra la información de un pedido junto con un botón de “**Finalizar compra**”. Cuando dicho botón es pulsado, se llama al endpoint “**pago/form**” del TPVV para así mostrar los detalles de la compra y el formulario para completar el pago.



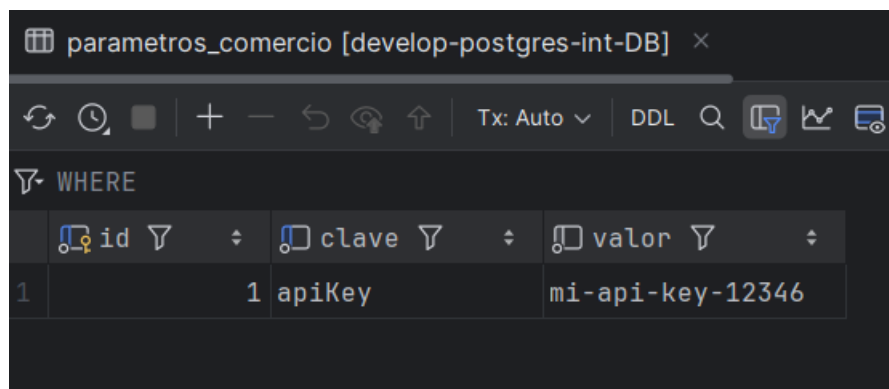
Figura 2: Información del pedido en el proyecto base que actúa como cliente

De esta manera, tenemos dos proyectos a considerar:

1. El proyecto **TPVV (plataforma de pago virtual)**, que opera como servidor y expone servicios de pasarela de pago.
2. El proyecto que simula una **Tienda Online**, que actúa como cliente y solicita los servicios de pago al servidor TPVV.

El proyecto del **TPVV** corre en el puerto **8123**, con una URL base de **http://localhost:8123/tpvv/boardalo**. En su base de datos se encuentran diversos comercios, cada uno con una **API Key** única.

El proyecto de la **Tienda Online** se ejecuta en el puerto **8246**, con una URL base de **http://localhost:8246**. Esta tienda está asociada a un comercio en concreto, por lo que dispone de la API Key que el TPVV requiere para validar la autenticación. Dicha Api Key se almacena en una tabla llamada “**parametros_comercio**”.



	id	clave	valor
1	1	apiKey	mi-api-key-12346

Figura 3: Tabla “parámetros_comercio”

La característica central de la integración radica en que la Tienda envía su **API Key** al TPVV mediante una comunicación **server to server**¹. Si la **API Key** concuerda con uno de los comercios registrados en la base de datos del TPVV, el proceso de pago se habilita. De lo contrario, se responde con un error 404.

Con la comunicación **server to server** se logra que la API Key no se exponga **nunca** en el navegador. De esta manera, la API Key no puede obtenerse nunca mediante una inspección del DOM. Esto resulta de vital importancia para evitar ataques de interceptación de información como el **Man-in-the-middle**².

¹ La comunicación **server-to-server** es un intercambio directo de datos entre dos servidores sin intervención de un cliente.

² El ataque **man-in-the-middle** es cuando un atacante intercepta y manipula la comunicación entre dos partes sin que ellas lo noten.

Mantener el encapsulamiento completo de la Api Key es un aspecto fundamental a la hora de mantener la confidencialidad y privacidad del comercio. La api key actúa como un identificador único del comercio y sirve para validar y autenticar su identidad.

El comercio podría estar utilizando la misma api key para realizar otros tipos de trámites diferentes al de realizar un pago. De esta forma, si se interceptase por un atacante la api key durante el proceso de un pago, la integridad del comercio se vería gravemente afectada. Esto es así puesto que esto no afectaría únicamente a la propia funcionalidad del pago, sino también al resto de funcionalidades que hiciesen uso de la api key, quedando el comercio en un estado de vulnerabilidad absoluta.

Para la implementación de la comunicación **server to server**, se ha hecho uso de **RestTemplate**, una clase de Spring Boot que facilita la comunicación HTTP entre servidores. Se ha configurado un bean³ de **RestTemplate** en la aplicación base de la tienda:

```
package interconnection.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;

@Configuration
public class AppConfig {

    /**
     * Bean de RestTemplate para realizar solicitudes HTTP.
     *
     * @return Una instancia de RestTemplate.
     */
    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

Figura 4: Bean de RestTemplate configurado en la aplicación base de la tienda

³ Un **Bean** en Spring Boot es un objeto gestionado por el **contenedor de Spring**. Representa un componente de la aplicación que Spring crea, configura y administra automáticamente, facilitando la inyección de dependencias y la modularidad del código.

3.3 OBJETIVOS DE LA INTERCONEXIÓN

1. **Brindar un formulario de pago** al usuario que compra en la Tienda, sirviéndose para ello del TPVV.
2. **Proteger el acceso** a los endpoints del TPVV mediante un mecanismo de autenticación basado en **API Keys**.
3. **Registrar y persistir** la información del pago, incluyendo tarjeta, importe y estado final (aceptado, rechazado o pendiente).
4. **Notificar a la Tienda** el resultado de cada transacción para que esta guarde en su propia base de datos los pedidos completados o pendientes.

3.4 DESCRIPCIÓN DE LA ARQUITECTURA EMPLEADA

La interconexión base que hemos implementado está compuesta por dos aplicaciones **Spring Boot** separadas, las cuales se comunican de forma **bidireccional** mediante una comunicación **server to server**:

1. **TPVV (Servidor)**
 - **URL Base:** `http://localhost:8123/tpvv/boardalo`
 - Mantiene una tabla de **Comercios** con sus respectivas **API Keys**.
 - Expone un **endpoint** para obtener el formulario de pago (`/pago/form`).
 - Expone un **endpoint** para procesar el pago (`/pago/realizar`).
 - Tras procesar un pago, **invoca** el endpoint de la Tienda para notificar la información del pedido completado.
2. **Tienda (Cliente)**
 - **URL Base:** `http://localhost:8246/`
 - Obtiene y muestra el formulario de pago que brinda el TPVV.
 - Envía las solicitudes de pago a **“/pago/realizar”** del TPVV.
 - Recibe la notificación final de pago (endpoint **/receivePedido**).
 - Almacena en su base de datos local el resultado del pedido.

3.5 DETALLES DE LA COMUNICACIÓN SERVER TO SERVER Y EL USO DE PROXIES

5.1 Visión Global del Flujo de Comunicación

1. **El usuario navega en la Tienda** y se muestra la información del pedido (mockeado)
2. **La Tienda** necesita mostrar un formulario de pago “**hosteado**” en el TPVV. Sin embargo, en lugar de redirigir directamente al TPVV, la Tienda actúa como un **proxy**⁴:
 - Llama al TPVV solicitando el HTML del formulario de pago a través del método “**pagoFormProxy**”.
 - Recibe ese HTML y lo **adapta** (inyecta campos ocultos, actualiza el <form>).
 - Entrega ese HTML final al navegador del usuario.

```
Optional<String> apiKeyOpt = parametroComercioService.getValorParametro( clave: "apiKey");
if (apiKeyOpt.isEmpty()) {
    model.addAttribute( attributeName: "error", attributeValue: "Error: API Key no encontrada.");
    return "error/404";
}
String apiKey = apiKeyOpt.get();

// Si hay errores en la query param, los mostramos
if (errors != null && !errors.isBlank()) {
    model.addAttribute( attributeName: "errorMessages", errors);
}

HttpHeaders headers = new HttpHeaders();
headers.set("Authorization", apiKey);

String url = "http://localhost:8123/tpvv/boardalo/pago/form?importe=" + precio
    + "&idTicket=" + ticket;

HttpEntity<String> entity = new HttpEntity<>(headers);

try {
    ResponseEntity<String> response =
        restTemplate.exchange(url, HttpMethod.GET, entity, String.class);
}
```

Figura 5: uso de RestTemplate para inyectar el HTML desde el servidor al cliente

3. **El usuario rellena los datos de la tarjeta** en la página servida por la Tienda y realiza el envío del formulario.

⁴ Un **proxy** es un intermediario que actúa entre un cliente y un servidor, gestionando las solicitudes y respuestas.

4. **De nuevo**, la petición de datos de la tarjeta (POST) se recibe primero en la Tienda, que vuelve a hacer una **llamada server to server** al TPVV para procesar el pago. Esto se realiza dentro del método “**realizarPagoProxy**”.

```
HttpEntity<PagoCompletoRequest> requestEntity = new HttpEntity<>(requestBody, headers);

String urlTPVV = "http://localhost:8123/tpvv/boardalo/pago/realizar";

try {

    ResponseEntity<String> response = restTemplate.postForEntity(urlTPVV, requestEntity, String.class);

    String body = response.getBody() != null ? response.getBody() : "";

}
```

Figura 6: la tienda usa **RestTemplate** para enviar al TPVV los datos del pago

5. **El TPVV** valida la **API Key**, registra el pago en su BD y, además, **notifica** a la Tienda que un pedido se ha procesado, con su estado final (rechazado, aceptado, pendiente). Todo esto se realiza dentro del método de **realizarPago**, dentro del fichero **PagoRestController.java** del proyecto del TPVV.

- Si el mensaje enviado desde el **TPVV** hasta la tienda empieza por “**OK|**”, entonces significa que el pago se ha registrado como un **pago aceptado**.
- Si el mensaje empieza por “**PEND|**”, entonces significa que el pago se ha registrado como un **pago pendiente**.
- Si el mensaje empieza por “**RECH|**”, entonces significa que el pago se ha registrado como un **pago pendiente**.
- Si el mensaje empieza por “**RECH|**”, entonces significa que el pago se ha registrado como un **pago rechazado**.
- Si el mensaje empieza por “**ERROR|**”, entonces significa que ha sucedido algún error a la hora de realizar el pago.

```
// Llamada POST al proyecto cliente, enviando pedidoCompletoRequest
Mono<String> response = webClient.post() RequestBodyUriSpec
    .uri(urlBack) RequestBodySpec
    .header(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON_VALUE)
    .body(BodyInserters.fromValue(pedidoCompletoRequest)) RequestHeadersSpec<capture of ?>
    .retrieve() ResponseSpec
    .bodyToMono(String.class);

String storeResponse = response.block();
```

Figura 7: El TPVV le envía a la tienda la información del pedido completado

6. En el caso del **TPVV**, para la comunicación **server to server** se ha utilizado **WebClient**, el cual a términos prácticos es equivalente a **RestTemplate**. Investigando, nos dimos que las dos formas principales de implementar comunicaciones **server to server** era usando tanto **WebClient** como **RestTemplate**, así que decidimos usar las dos, una para el **TPVV** y otra para la **tienda base**. Así porbábamos ambas alternativas.

```
© WebClientConfig.java x
1 package tpvv.config;
2
3 import org.springframework.context.annotation.Bean;
4 import org.springframework.context.annotation.Configuration;
5 import org.springframework.web.reactive.function.client.WebClient;
6
7 @Configuration  Cristian Andrés Córdoba Silvestre
8 public class WebClientConfig {
9
10     @Bean  Cristian Andrés Córdoba Silvestre
11     > public WebClient webClient() { return WebClient.builder().build(); }
14 }
15
```

Figura 8: Configuración del **Bean** de **WebClient** dentro del proyecto del **TPVV**


```

    if (estadoPago.startsWith("RECH")) {
        return ResponseEntity.ok( body: "RECH|" + razonEstadoPago + ".");
    }

    else if (estadoPago.startsWith("PEND")) {
        return ResponseEntity.ok( body: "PEND|" + razonEstadoPago + ".");
    }

    else {
        return ResponseEntity.ok( body: "OK|" + razonEstadoPago + ".");
    }

} catch (IllegalArgumentException ex) {

    return ResponseEntity.ok( body: "ERROR|Error 404");
}
}

```

Figura 9: Manejo del estado del pago en la respuesta desde el TPVV a la tienda

7. Finalmente, la tienda recibe el mensaje correspondiente del **TPVV** y registra en su BD la información del nuevo pago y del nuevo pedido completado.

```

@PostMapping("/receivePedido")  Cristian Andrés Córdoba Silvestre
public ResponseEntity<String> receivePedido(@RequestBody PedidoCompletoRequest request) {
    log.debug("Recibido en la tienda un PedidoCompletoRequest: {}", request);

    try {
        pagoService.procesarPedido(request);
        return ResponseEntity.ok( body: "Pedido recibido y guardado con éxito.");
    } catch (IllegalArgumentException ex) {
        return ResponseEntity.badRequest().body("Error 404");
    }
}
}

```

Figura 10: La tienda guarda la información del nuevo pedido completado en la BD

3.6 CLASES Y COMPONENTES RELEVANTES

3.6.1. En el TPVV

1. ApiKeyAuthFilter y SecurityConfig

- Se encargan de extraer la cabecera Authorization y de validar la **API Key**. Si la API Key no coincide con la de ningún comercio registrado en la BD del TPVV, entonces se devuelve un error **404**.
- Configuran que solo rutas /pago/** requieran la autenticación por medio de la **API Key** en la cabecera.

```
// Validar API-Key solo en rutas protegidas (ej: /pago/**)
if (path.startsWith("/pago/")) {
    String apiKey = extractApiKey(request);

    if (StringUtils.hasText(apiKey)) {
        Optional<Comercio> comercioOpt = comercioRepository.findByApiKey(apiKey);
        if (comercioOpt.isPresent()) {
            Comercio comercio = comercioOpt.get();
            UsernamePasswordAuthenticationToken authentication = new UsernamePasswordAuthenticationToken(
                comercio, credentials: null, Collections.emptyList());
            authentication.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
            SecurityContextHolder.getContext().setAuthentication(authentication);
        } else {
            // API-Key no válida
            response.sendError(HttpServletResponse.SC_NOT_FOUND);
            return;
        }
    } else {
        // Falta la API-Key en la cabecera Authorization
        response.sendError(HttpServletResponse.SC_NOT_FOUND);
        return;
    }
}

// Continuar con el filtro
filterChain.doFilter(request, response);
}
```

Figura 11: fichero ApiKeyAuthFilter.java

```

// Permitir iframes en la misma ruta (necesario para la consola H2)
.headers( HeadersConfigurer<HttpSecurity> headers -> headers.frameOptions( FrameOptionsConfig frame -> frame.sameOrigin()))

// Control de sesiones (STATELESS)
.sessionManagement( SessionManagementConfigurer<HttpSecurity> session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))

// Reglas de autorización
.authorizeHttpRequests( AuthorizationManagerRequestMat... auth -> auth
    .requestMatchers( @"/h2-console/**").permitAll() // Permitir H2
    .requestMatchers( @"/pago/**").authenticated() // Exigir autenticación en /pago
    .anyRequest().permitAll() // Todo lo demás, libre
)

// Filtro personalizado antes de UsernamePasswordAuthenticationFilter
.addFilterBefore(apiKeyAuthFilter, UsernamePasswordAuthenticationFilter.class)

// Autenticación básica
.httpBasic(Customizer.withDefaults());

// Construye el SecurityFilterChain
return http.build();

```

Figura 12: Fichero SecurityConfig.java

2. PagoController (Controlador web con Thymeleaf)

- Método mostrarFormularioPago(...): Muestra la plantilla paymentForm.html.

3. PagoRestController (Controlador REST)

- Método realizarPago(...): Procesa el JSON del pago, valida, guarda y notifica a la Tienda.

4. PagoService

- Lógica de negocio principal para persistir el pago, determinar el estado y crear el DTO PedidoCompletoRequest que enviará de vuelta a la Tienda.

3.6.2. En la Tienda

1. StoreController

- pagoFormProxy(...): Hace GET al TPVV para obtener el formulario, actúa como proxy inyectando datos.
- realizarPagoProxy(...): Hace POST al TPVV para realizar el pago y luego interpreta la respuesta (OK|, RECH|, PEND|).

2. PagoService (en la Tienda)

- Método procesarPedido(...): Recibe el PedidoCompletoRequest proveniente del TPVV y lo almacena en la tabla pedido_completados.

3. Entidades: PedidoCompletado, ParametroComercio, etc.

- Permiten almacenar la configuración (por ejemplo, la apiKey en la tabla parametros_comercio) y los registros de pedidos finalizados.

4. **RestTemplate** y su configuración en **AppConfig.java**.

- Para orquestar las llamadas HTTP desde la Tienda al TPVV.

3.7 RESUMEN DE LOS BENEFICIOS Y MOTIVACIÓN DE LA COMUNICACIÓN SERVER TO SERVER Y DEL USO DE PROXIES

- **Control Centralizado del Flujo:** La Tienda, al hacer de proxy, conserva el control de la experiencia de usuario, sin exponer directamente la URL del TPVV.
- **Seguridad:** Se evita que el usuario interactúe directamente con la plataforma de pago, reduciendo el riesgo de exponer en el navegador la API Key o datos de implementación interna.
- **Mantenimiento:** Cambios futuros en la plataforma de pago pueden ser transparentes para el usuario, siempre y cuando la Tienda se encargue de “inyectar” la estructura de HTML que corresponda.

3.8 CONSIDERACIONES ADICIONALES

Para el formulario dentro de la pasarela de pago, se han realizado las siguientes validaciones:

- Ninguno de los campos puede estar vacío.
- El número de tarjeta ha de contener **exactamente** 16 dígitos. Los dígitos pueden escribirse o bien todos juntos o bien separados por un único espacio por cada grupo de cuatro dígitos (ejemplos de números de tarjeta válidos serían: “1234123412341234” y también “1234 1234 1234 1234”).
- La fecha de caducidad debe de seguir **exactamente** el formato “mm/aa”.
- El código de seguridad ha de contener **exactamente** 3 dígitos.

Cuando falla cualquiera de las validaciones mencionadas, se muestra por pantalla los mensajes de error correspondientes.

Errores en los datos introducidos: El nombre del titular de la tarjeta no puede estar vacío. El número de tarjeta no puede estar vacío. La fecha de caducidad no puede estar vacía. El código de seguridad no puede estar vacío.

The screenshot shows a payment form titled "TPVV BoarDalo". It is divided into two main sections: "Datos de la Compra" (Purchase Data) and "Pago con tarjeta" (Card Payment). The "Datos de la Compra" section contains the following information: Importe: 888.89 €, Comercio: Tienda Online v2, Nº de pedido: TICKET-888, Fecha: 09/09/2029, and Hora: 13:45. The "Pago con tarjeta" section contains the following fields: Nombre Completo (empty), Nº Tarjeta: (empty, with a placeholder "0000 0000 0000 0000"), Caducidad (empty, with a placeholder "mm/aa"), and Cód. Seguridad (empty, with a placeholder "xxx"). An "Aceptar" button is located at the bottom right of the "Pago con tarjeta" section.

Figura 13: Error de validación. Todos los campos están vacíos.

Errores en los datos introducidos: El número de tarjeta debe tener exactamente 16 dígitos (los dígitos deben de introducirse todos seguidos o bien dejando un ÚNICO espacio en blanco entre cada grupo de cuatro dígitos). La fecha de caducidad debe tener el formato mm/yy. El CVC debe tener exactamente 3 dígitos.

The screenshot shows the same payment form as in Figure 13, but with different data entered. The "Datos de la Compra" section remains the same. The "Pago con tarjeta" section contains the following fields: Nombre Completo (empty), Nº Tarjeta: (empty, with a placeholder "0000 0000 0000 0000"), Caducidad (empty, with a placeholder "mm/aa"), and Cód. Seguridad (empty, with a placeholder "xxx"). An "Aceptar" button is located at the bottom right of the "Pago con tarjeta" section.

Figura 14: Error de validación. Se han introducido los datos con un formato incorrecto

Para realizar todas las validaciones se ha empleado regex⁵.

```
if (tarjetaData.getNumeroTarjeta() == null || tarjetaData.getNumeroTarjeta().isBlank()) {
    errorMsg.append("El número de tarjeta no puede estar vacío. ");
} else if (!tarjetaData.getNumeroTarjeta().matches("^(\\d{16}|\\d{4} \\d{4} \\d{4} \\d{4})$")) {
    errorMsg.append("El número de tarjeta debe tener exactamente 16 dígitos (los dígitos deben de int
}

if (tarjetaData.getFechaCaducidad() == null || tarjetaData.getFechaCaducidad().isBlank()) {
    errorMsg.append("La fecha de caducidad no puede estar vacía. ");
} else if (!tarjetaData.getFechaCaducidad().matches("(0[1-9]|1[0-2])\\/(\\d{2})$")) {
    errorMsg.append("La fecha de caducidad debe tener el formato mm/yy. ");
}

if (tarjetaData.getCvc() == null || tarjetaData.getCvc().isBlank()) {
    errorMsg.append("El código de seguridad no puede estar vacío. ");
} else if (!tarjetaData.getCvc().matches("^\\d{3}$")) {
    errorMsg.append("El CVC debe tener exactamente 3 dígitos. ");
}
```

Figura 15: Uso de regex en el proyecto del TPVV para realizar las validaciones en los datos del pago

En base a los cuatro primeros dígitos del número de la tarjeta, se guarda un pago con uno u otro estado de pago de la siguiente manera:

0000 **** * → RECH0001

0001 **** * → RECH0002

0002 **** * → RECH0003

0003 **** * → RECH0004

1000 **** * → PEND0001

1001 **** * → PEND0002

1002 **** * → PEND0003

1003 **** * → PEND0004

2000 **** * → ACCEPT0001

2001 **** * → ACCEPT0002

2002 **** * → ACCEPT0003

2003 **** * → ACCEPT0004

⁵ **regex** (expresión regular) es un patrón de texto utilizado para buscar, coincidir o validar cadenas. Es ampliamente utilizado en validaciones para asegurarse de que los datos cumplan ciertos formatos, como correos electrónicos o contraseñas.

Cualquier otro número de tarjeta → **ACEPT1000**

WHERE

	id	nombre	razon_estado
1	1	ACEPT1000	PAGO ACEPTADO: PAGO PROCESADO...
2	2	ACEPT0001	PAGO ACEPTADO: IDENTIDAD DEL ...
3	3	ACEPT0002	PAGO ACEPTADO: REVISIÓN ANTIF...
4	4	ACEPT0003	PAGO ACEPTADO: CONFIRMACIÓN I...
5	5	ACEPT0004	PAGO ACEPTADO: MONEDA SOPORTA...
6	6	PEND0001	PAGO PENDIENTE: VERIFICACIÓN ...
7	7	PEND0002	PAGO PENDIENTE: TRANSFERENCIA...
8	8	PEND0003	PAGO PENDIENTE: CONVERSIÓN DE...
9	9	PEND0004	PAGO PENDIENTE: PROCESO DE CO...
10	10	RECH0001	PAGO RECHAZADO: SALDO INSUFIC...
11	11	RECH0002	PAGO RECHAZADO: TARJETA BLOQU...
12	12	RECH0003	PAGO RECHAZADO: TARJETA VENCI...
13	13	RECH0004	PAGO RECHAZADO: FALLO EN LA C...

Figura 16: Tipos de Estado de Pago guardados en la BD

De esta forma , dependiendo del tipo de estado de pago, al recibir los datos del pedido completado, la tienda mostrará un tipo diferente de pantalla:

Pago rechazado

Razón del estado

PAGO RECHAZADO: SALDO INSUFICIENTE.

[Volver a la información del pedido](#)

Figura 17: Pantalla mostrada en la tienda tras haberse efectuado un pago rechazado

Pago Pendiente

Razón del estado

PAGO PENDIENTE: TRANSFERENCIA EN ESPERA DE COMPENSACIÓN.

[Volver a la información del pedido](#)

Figura 18: Pantalla mostrada en la tienda tras haberse efectuado un pago pendiente

Pago Aceptado

Razón del estado

PAGO ACEPTADO: PAGO PROCESADO CORRECTAMENTE.

[Volver a la información del pedido](#)

Figura 19: Pantalla mostrada en la tienda tras haberse efectuado un pago rechazado

3.9. CONCLUSIONES

La integración entre la **Tienda** y el **TPVV** demuestra una forma robusta de implementar un **flujo de pago**. La **API Key** provee un mecanismo simple de autenticación para cada comercio, y la **comunicación server to server** garantiza que los datos de pago se procesen y almacenen en la plataforma de forma segura, a la vez que la **Tienda** conserva la información final de los pedidos, proporcionando así una solución modular, escalable y fácil de mantener.

En este planteamiento, la Tienda hace un uso intensivo de **llamadas proxy**:

1. **Para el formulario** (GET al TPVV)
2. **Para el procesamiento** (POST al TPVV)

Por su parte, el TPVV complementa esta interacción **notificando** a la Tienda los resultados. Esta arquitectura ofrece varias ventajas en términos de **control**, **seguridad** y **mantenibilidad** de la solución. Aún así, existen diferentes aspectos que podrían mejorarse en implementaciones futuras.

Puntos de Mejora:

- **Almacenamiento seguro** de la **API Key** en la Tienda, evitando exponerla en texto plano.
- Reducir la manipulación de HTML por “reemplazos de cadena” e implementar soluciones más escalables (p. ej., exponer un JSON con la información del formulario y que la Tienda renderice su propio template).
- Incluir un sistema de **firmas HMAC** o **JWT** para robustecer la seguridad de la comunicación más allá de la simple verificación de API Key.

No obstante, el aspecto más importante a mejorar a tener en cuenta para implementaciones futuras sería el aseguramiento de que los datos del pago **no cambien** durante el propio proceso de realización del pago.

Básicamente con esto se pretende evitar que, por ejemplo, un atacante llegue a la pantalla de pasarela de pago, cambie el importe, (por ejemplo a través del DOM), y se termine efectuando un pago con un importe mucho menor al que tenía el pago real

Evidentemente, esto representa una acción realmente peligrosa, ya que cualquiera podría realizar cambios en los datos del pago desde la pantalla de la pasarela de pago (simplemente bastaría con acceder al **DOM** y cambiar desde allí cualquier elemento del HTML y después darle a realizar pago).

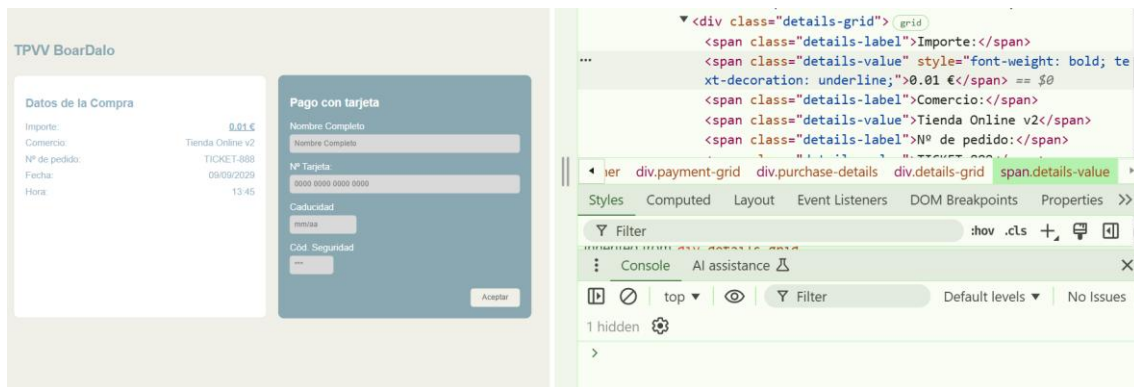


Figura 20: Cambiando los elementos del pago directamente desde el DOM

Para solucionar este problema, llegados a dos posibles opciones:

1. Implementar un sistema de **firmas**, de forma que cuando la tienda acceda a la pasarela de pago, el mismo **backend** de la tienda se encargue de cifrar mediante una clave pública los datos iniciales del pago (**importe, ticket**). Estos datos cifrados serían enviados al TPVV y almacenados en su BD. Cuando la tienda completase el pago, estos datos iniciales volverían a enviarse cifrados con la clave pública junto con el resto de información del pago. Cuando el **TPVV** reciba estos datos, si los datos cifrados iniciales no coinciden con los datos cifrados iniciales, entonces esto es un indicio de que ha sucedido alguna acción sospechosa, con lo que no se completaría el pago y se lanzaría un mensaje de error **404**.
2. Cuando la tienda acceda a la pasarela de pago, los datos iniciales del pago (**importe, ticket**) son enviados tal cual al **TPVV**. El **TPVV** recibe estos datos y los almacena en una tabla temporal. Una vez que la tienda haya completado el pago, todos los datos del pago son enviados al **TPVV**. Cuando el **TPVV** recibe estos datos, los compara con los datos iniciales almacenados en la tabla temporal. Si no coinciden, entonces el pago no se completa y se lanza un mensaje de error **404**.

Debido al alcance del proyecto y a las limitaciones temporales del mismo, al final no terminamos estas consideraciones. No obstante, es fundamental que este requisito fuese implementado en actualizaciones futuras de la aplicación.