



Universidad de Valladolid

Escuela Técnica Superior de Ingenieros de Telecomunicación

Trabajo de Fin de Grado

GRADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

**Request To Pay frente a la domiciliación bancaria:
propuesta de mejora e implementación de un prototipo.**

Autor:

Alonso Sandoval Martínez

Tutor:

Federico Simmross Wattenberg

Valladolid, junio 2025

Índice

Agradecimientos	2
Resumen	3
1 Introducción	4
1.1 Motivación	4
1.1.1 Ineficiencias operativas detectadas	4
1.1.2 Oportunidad de un esquema Request-to-Pay	7
1.2 Objetivos	8
1.3 Fases y Métodos	9
1.4 Medios necesarios empleados para el desarrollo	9
2 Antecedentes y estado del arte	10
2.1 Evolución de los medios de pago hacia SEPA	10
2.2 El papel del EPC en la estandarización y el surgimiento de Request-to-Pay	10
2.3 Funcionamiento técnico de SEPA Request-to-Pay (SRTP)	11
2.4 RTP dentro del ecosistema de pagos: arquitectura en capas	20
3 Diseño e Implementación	26
3.1 Fundamentos Teóricos	26
3.2 Tecnologías utilizadas	27
3.3 Estructura y funcionamiento	28
3.3.1 Backend	28
3.3.2 Frontend	34
3.4 Emulación del prototipo <i>Request To Pay</i>	37
3.5 Pruebas y validación	48
4 Conclusiones y líneas futuras	55
4.1 Potencial y aplicación real	55
Índice de Figuras	58
Índice de Tablas	59
Índice de Códigos	60

Agradecimientos

Agradezco, en primer lugar, la orientación y el seguimiento de mi tutor, cuya experiencia ha sido decisiva para completar este trabajo.

Extiendo mi gratitud a mi familia por su respaldo constante y a mis amigos por el apoyo práctico y la paciencia mostrada durante el desarrollo del proyecto.

Su ayuda conjunta ha permitido que este trabajo llegue a buen término.

Resumen

Este Trabajo de Fin de Grado (TFG) explora cómo mejorar los sistemas de cobro en Europa, comparando el método tradicional de domiciliación bancaria (SEPA Direct Debit o SDD) con una solución más moderna: el esquema Request To Pay (RTP). El SDD, aunque muy usado, tiene limitaciones como procesos lentos, necesidad de documentos físicos y riesgos de devoluciones, lo que lo hace poco práctico para el mundo digital actual, donde se busca rapidez y simplicidad. En cambio, el RTP permite a quien cobra enviar una solicitud de pago digital que el pagador puede aceptar o rechazar al instante, haciendo el proceso más rápido, seguro y eficiente.

El objetivo del TFG ha sido crear un prototipo que simule cómo funcionaría un proveedor de RTP. Este prototipo, desarrollado con herramientas como Node.js y una base de datos sencilla, muestra cómo se pueden gestionar solicitudes de pago en tiempo real, desde su creación hasta su aprobación o rechazo. Las pruebas realizadas confirman que el sistema funciona bien y resuelve problemas del SDD, como la lentitud y la falta de control inmediato.

Este trabajo no solo demuestra que el RTP puede ser una alternativa útil para modernizar los pagos, sino que también abre la puerta a mejoras futuras, como hacerlo más seguro o conectarlo con bancos reales. En resumen, el prototipo es un paso hacia un sistema de cobros más ágil y adaptado a las necesidades de hoy, con potencial para cambiar cómo manejamos las transacciones en Europa.

1 Introducción

Para comprender el entorno actual de los pagos en Europa, conviene arrancar por la *Single Euro Payments Area* (SEPA): un espacio comunitario en el que todos los pagos en euros se rigen por los mismos estándares técnicos y normas operativas, de modo que enviar dinero de un país a otro resulta tan ágil y claro como una transferencia nacional. SEPA estableció protocolos de mensajería comunes, armonizó los plazos de liquidación y fijó reglas uniformes de protección al usuario, creando la base sobre la que se despliegan hoy los servicios de pago más innovadores.

En los últimos diez años, la digitalización de los servicios financieros ha cambiado por completo cómo particulares y empresas gestionan sus transacciones dentro de ese marco SEPA. Las transferencias instantáneas, las API abiertas de los bancos y el auge del comercio electrónico han disparado la demanda de procesos de cobro que sean sencillos, transparentes y en tiempo real. No obstante, los instrumentos de pago tradicionales —tarjetas, transferencias convencionales o domiciliaciones— nacieron en un contexto muy distinto y todavía arrastran limitaciones que penalizan tanto la experiencia de usuario como la eficiencia operativa.

Aquí es donde entra en juego *Request-to-Pay* (RTP). Este servicio de mensajería permite al beneficiario enviar al pagador una solicitud de pago digital estructurada, con todos los detalles (importe, concepto, vencimiento), y recibir en segundos una respuesta —aceptación, rechazo o aplazamiento— antes de iniciar el movimiento de fondos. RTP no sustituye los métodos de pago existentes, sino que actúa como una capa de orquestación sobre la infraestructura SEPA (y, en especial, los pagos inmediatos) y los canales de banca online, facilitando la conciliación, reduciendo la fricción en el cobro y modernizando la experiencia tanto para empresas como para consumidores.

1.1 Motivación

La domiciliación bancaria regulada por el esquema *SEPA Direct Debit* (SDD)¹ desde 2014, sigue siendo el método principal para cobros recurrentes en España. No obstante, su estructura, pensada para un entorno de procesos *offline*—genera hoy inconvenientes que chocan con las demandas de inmediatez, seguridad y experiencia de usuario fluida que caracterizan la economía digital actual.

1.1.1 Ineficiencias operativas detectadas

Tras analizar la operativa SDD nacional se han identificado una serie de ineficiencias que afectan tanto a los usuarios como a las entidades participantes en el proceso de pago resumidas en los siguientes 5 puntos:

1. Modelo off-line y necesidad de mandato físico

El proceso de pago mediante el esquema SDD opera bajo un modelo offline, lo que implica una ausencia total de interacción en tiempo real entre las partes involucradas. Para iniciar el cobro, el deudor debe firmar y enviar un **mandato SEPA**² en formato físico. Este documento debe ser conservado por el acreedor durante toda la duración del contrato y hasta 14 meses después de la última transacción realizada. Aunque la digitalización ha avanzado en muchos ámbitos, aún no existe un estándar único y interoperable para los **eMandates**³, lo que lleva

¹ *SEPA Direct Debit* es el instrumento paneuropeo de cargo en cuenta regulado por el *European Payments Council*.

² El mandato SEPA es un documento mediante el cual el deudor autoriza al acreedor a realizar cobros automáticos a través de la domiciliación bancaria.

³ Un eMandate es la versión electrónica del mandato SEPA, que permite autorizar cobros de manera digital.

a que cada entidad bancaria implemente su propio sistema. Esta falta de uniformidad genera inconsistencias y dificulta la estandarización del proceso. **Consecuencias:**

- Fricciones significativas en los procesos de venta digital, ya que los usuarios deben completar pasos adicionales que rompen con la inmediatez esperada en el comercio electrónico actual.
- Costes operativos considerables asociados a la gestión administrativa, como el archivado, las auditorías y el mantenimiento de los mandatos físicos.
- Riesgos legales y financieros en caso de disputa, como devoluciones costosas o conflictos prolongados con los deudores, debido a la ausencia de un mandato válido.

2. Derecho a devolución prolongado

El esquema SDD otorga al deudor un derecho a devolución excepcionalmente amplio, lo que genera incertidumbre en la gestión de los ingresos por parte de los acreedores. En el caso de un *cobro autorizado*, el deudor puede solicitar la devolución del importe sin necesidad de justificar su decisión durante un periodo de **ocho semanas**, bajo la política conocida como “*no-questions-asked*”. Por otro lado, si el cobro se clasifica como *no autorizado* —por ejemplo, si el banco emisor no puede probar la existencia de un mandato válido—, el plazo para reclamar se extiende hasta **trece meses**.

Consecuencias:

- Notable inseguridad para los acreedores, quienes deben mantener reservas de liquidez y provisiones contables para cubrir posibles devoluciones tardías.
- Facilitación de prácticas como el *friendly fraud*⁴, donde los deudores reclaman reembolsos injustificados tras haber recibido el producto o servicio.
- Impacto directo en la rentabilidad de las empresas debido a las devoluciones inesperadas.

3. Ciclos de cobro lentos

Los tiempos de procesamiento en el esquema SDD son significativamente prolongados, lo que compromete tanto la eficiencia operativa como la experiencia del usuario. En el esquema CORE⁵, el acreedor debe enviar la orden de cobro al banco con una antelación de **D-5 días** para la primera domiciliación y de **D-2 días** para las domiciliaciones recurrentes. A esto se suman **dos días adicionales** para la liquidación interbancaria. En total, el proceso puede demorar entre seis y ocho días naturales desde que se solicita el cobro hasta que se confirma el abono, un plazo incompatible con las expectativas de inmediatez en la venta de bienes o servicios digitales.

Consecuencias:

- Afectación en la planificación financiera de las empresas, ya que los ingresos no están disponibles de manera inmediata, generando una tesorería imprevisible.
- Riesgo de prestar servicios o entregar productos sin la certeza de que el pago se completará con éxito.

⁴El *friendly fraud* ocurre cuando un usuario consume un bien o servicio y, posteriormente, solicita una devolución sin justificación, aprovechando las políticas de devolución laxas.

⁵El esquema CORE es el estándar de domiciliación bancaria SEPA utilizado para pagos entre empresas y consumidores.

- Pérdidas económicas significativas debido a la falta de confirmación inmediata del pago.

4. Costes y complejidad de las R-transactions⁶

Las **R-transactions** representan una fuente notable de complicaciones y costes adicionales. Estas transacciones se clasifican mediante diversos códigos, cada uno asociado a un flujo y reglas específicas, lo que dificulta su gestión y seguimiento. Los acreedores deben dedicar recursos a identificar las causas de cada R-transaction y aplicar las medidas correctivas correspondientes, un proceso que frecuentemente requiere intervención manual debido a la falta de automatización.

Consecuencias:

- Necesidad de equipos especializados en conciliación y recobro, incrementando los costes operativos.
- Reducción de la eficiencia general del sistema debido a la complejidad de los procesos.
- Posibilidad de errores o retrasos que afectan la productividad y la confianza en el esquema SDD.

5. Ausencia de autorización fuerte (SCA)⁷

El esquema SDD se basa en un consentimiento previo otorgado mediante el mandato SEPA, pero no incorpora la **autorización fuerte del cliente (SCA)** en el momento de cada transacción. Una vez firmado el mandato, los cobros se ejecutan automáticamente sin que se solicite al deudor una autenticación adicional para cada operación.

Consecuencias:

- Elevado riesgo de disputas por cargos no autorizados, lo que puede derivar en conflictos y devoluciones.
- Pérdida de una oportunidad clave para fortalecer la seguridad y la confianza en el proceso de cobro mediante métodos de autenticación modernos.

En conclusión. Estas ineficiencias tienen un gran impacto en la operativa y la competitividad de las empresas y entidades financieras que lo utilizan y se traducen en:

- Una **estructura de costes elevada**, derivada de la alta frecuencia de devoluciones y la necesidad de personal especializado para gestionarlas, lo que incrementa los gastos operativos.
- **Liquidez incierta**, ya que los ingresos no se confirman de inmediato y pueden ser revertidos incluso meses después de haberse registrado, dificultando la gestión financiera.
- Un **freno al desarrollo de la economía digital**, puesto que el SDD no está diseñado para ofrecer experiencias de pago instantáneas y fluidas, como las que proporcionan métodos alternativos como las tarjetas de crédito, los monederos electrónicos o plataformas como Bizum.

⁶Transacciones de rechazo, devolución o reembolso asociadas a pagos fallidos o no autorizados.

⁷La SCA (Strong Customer Authentication) es un requisito de seguridad establecido por la directiva PSD2, que exige la verificación del usuario mediante al menos dos factores de autenticación.

1.1.2 Oportunidad de un esquema Request-to-Pay

El estándar *SEPA Request-to-Pay* (**SRTP**)⁸ aborda de manera efectiva las limitaciones técnicas y operativas del SDD, ofreciendo una laternaativa más ágil y adaptada al entorno digital.

A continuación se describen las principales ventajas del SRTP frente al SDD:

a) **Autenticación reforzada y consentimiento digital inmediato**

El SRTP reemplaza el mandato físico del SDD por una solicitud de pago que el deudor aprueba directamente desde su banca en línea o wallet digital mediante SCA. Este proceso genera una prueba electrónica de consentimiento, firmada y resgistrada en el sistema del PSP del pagador, eliminando la dependencia de documentos en papel y simplificando la gestión de autorizaciones.

b) **Irrevocabilidad y mitigación de fraude *post-servicio***

Una vez aceptada la solicitud, el pago se ejecuta mediante SCT Inst⁹. A diferencia del SDD que permite devoluciones automáticas en plazos amplios, el SRTP no admite reversiones sin causa justificada. Esto minimiza el riesgo de textbffaude amistoso-donde el deudor reclama devoluciones tras recibir un servicio- y reduce la necesidad de provisiones por impagos.

c) **Liquidez *real-time* y conciliación automática**

Con fondos disponibles en menos de 10 segundos, las empresas pueden gestionar su tesorería con mayor precisión. Además, el uso de identificadores únicos y referencias estructuradas según el estándar ISO 20022 asegura que la información del pago se transmita íntegramente de extremo a extremo, permitiendo una conciliación automática y eliminando los retrasos y errores típicos del SDD.

d) **Simplificación operativa**

El SRTP elimina las R-transactions, la custodia de mandatos físicos y las tareas administrativas asociadas. El flujo se reduce a dos mensajes principales -solicitud y aceptación-, con la opción de una transferencia instantánea, ofreciendo una trazabilidad clara y directa.

e) **Flexibilidad comercial y costes reducidos**

Este esquema soporta cobros únicos, recurrentes o fraccionados a través de canales digitales como enlaces profundos, códigos QR o APIs. Al estar basado en SCT Inst, las comisiones bancarias son bastante menores a las de las tarjetas o la destión de devoluciones del SDD, lo que mejora la eficiencias y amplía su aplicabilidad en el comercio electrónico.

En conjunto, el SRTP conserva los puntos fuertes del SDD pero los adapta a las necesidades actuales, proporcionando una solución más rápida, segura y eficiente. Al superar las ineficiencias del SDD se convierte en una herramienta clave para modernizar los sistema de pago en la zona SEPA y, en concreto, en España.

⁸Iniciativa del European Payments Council que define un flujo de solicitud (*request*) y aceptación de pagos en tiempo real, apoyado en mensajería **ISO 20022** y siendo independiente del instrumento de liquidación posterior.

⁹SCT INST: transferencia inmediata SEPA con liquidación en menos de 10 s.

Cuadro 1: Comparativa entre SDD y SRTP con SCT Inst

Aspecto	SDD	SRTP (+ SCT Inst)
Autorización	Mandato off-line	Consentimiento digital (SCA)
Plazo de devolución	8 semanas / 13 meses	No aplica (irrevocable)
Disponibilidad de fondos	5–8 días	Menos de 10 segundos
Coste operativo	Alto (mandatos, R-CODES)	Bajo (mensajería ISO 20022)
Cobertura <i>e-commerce</i>	Limitada	Amplia (API / móvil)
Riesgo de fraude	Medio-Alto (devoluciones)	Bajo (SCA + irreversibilidad)

1.2 Objetivos

El propósito central de este TFG es desarrollar un sistema de software que simule, de principio a fin, un proveedor del esquema SRTP. Este proyecto nace con la idea de abordar las limitaciones del SDD, explicadas anteriormente. Se quiere demostrar que el SRTP puede ser una solución moderna, ágil y segura para los pagos en Europa, alineada con la versión 4.0 del *SRTP Scheme Rulebook* [Cou25] y las guías técnicas del European Payments Council (*EPC137* y *EPC164*) [Cou23a, Cou23b].

El sistema será un prototipo funcional que se pueda instalar fácilmente con **Docker Compose** y que ofrezca una API HTTP/JSON basada en el estándar *OpenAPI 3.1*. Este prototipo debe cubrir las cuatro operaciones principales del flujo SRTP: crear una solicitud de pago (*create*), rechazarla (*reject*), responder a ella (*response*) y cancelarla (*cancel*). En resumen, queremos construir una herramienta que no solo funcione, sino que también sea práctica y fiel a las especificaciones del SRTP.

Para mantener el proyecto enfocado y medible, hemos definido los siguientes **objetivos específicos**:

1. Desarrollar una API robusta y eficiente.

Implementar, usando **Node.js 20 LTS** y **Express**, un conjunto de *endpoints* REST que garanticen alta disponibilidad. Además, incluir un sistema de notificaciones asíncronas mediante **Socket.IO** para que las actualizaciones lleguen en tiempo real, reflejando la inmediatez que el SRTP promete frente al lento SDD.

2. Garantizar seguridad y confianza.

Crear un módulo que gestione firmas digitales, sellado temporal y validación con certificados X.509 (*QSeal/QWAC*). Esto asegurará que el sistema cumpla con los requisitos de identificación, autenticación y autorización del *API Security Framework* [Cou23b], protegiendo cada transacción y resolviendo la falta de autenticación fuerte del SDD.

3. Almacenar datos de forma sencilla y fiable.

Diseñar una base de datos ligera en **SQLite**, gestionada con **SQLAlchemy**, para guardar el estado de las operaciones y su auditoría. El modelo de datos estará alineado con los *datasets* DS-02, DS-07 y DS-10 del *Rulebook*, asegurando que todo esté bien organizado y traceable.

4. Asegurar la calidad con pruebas automatizadas.

Entregar una colección de pruebas en **Postman** y un flujo de integración continua en **GitHub**

Actions. Estas pruebas verificarán la integración del sistema y validarán los esquemas JSON contra los estándares oficiales del EPC, garantizando que el prototipo funcione como se espera.

5. Documentar y planificar mejoras futuras.

Identificar cualquier diferencia entre las especificaciones del SRTP y nuestra implementación, explicando por qué ocurrieron. También propondremos una hoja de ruta clara para ajustar el sistema y hacerlo compatible con el *Electronic Data Submission* (EDS) en el futuro, asegurando su relevancia a largo plazo.

En definitiva, este TFG busca construir una solución práctica que demuestre el potencial del SRTP para superar las trabas del SDD, usando tecnologías modernas y un enfoque riguroso. Queremos que este prototipo no solo cumpla con los estándares técnicos, sino que también inspire confianza en una nueva forma de gestionar pagos en Europa.

1.3 Fases y Métodos

El TFG se ha estructurado en 3 fases principales:

Fase 1 – Análisis y planificación En esta primera etapa se estudió el mundo de los pagos en la zona SEPA, revisando los documentos emitidos por el EPC para identificar las posibles mejoras que el RTP podría suponer.

Luego, se estudiaron los casos de uso del RTP, identificando qué necesitan hacer los actores principales y se planificó el prototipo.

Fase 2 – Diseño e implementación Una vez claro el contexto, se comenzó a diseñar la estructura del prototipo definiendo los roles de los actores y cómo interactúan entre sí.

La implementación se ha llevado a cabo usando herramientas que se detallarán posteriormente.

Fase 3 – Pruebas y validación Por último, una vez implementado el prototipo se realizaron una serie de pruebas y comprobaciones para verificar que todo funciona correctamente, como veremos en otro apartado del documento.

1.4 Medios necesarios empleados para el desarrollo

- **Software de desarrollo:** Node.js 20 LTS, Express 4, Socket.IO 4, Sequelize 6, Jest, Postman v10, Docker 24, Docker Compose v2, Git y GitHub Actions.
- **Herramientas de apoyo:** OpenSSL 3 para gestión de certificados, toxiproxy para pruebas de resiliencia, Spectral OCI para linting de especificaciones OpenAPI.
- **Documentación oficial:** SRTP Scheme Rulebook v4.0 [Cou25], SRTP related API Specifications v3.1 [Cou23a], API Security Framework v2.0 [Cou23b], ISO 20022 *pacs/pain/camt*, directivas PSD2/eIDAS.
- **Hardware y S.O.:** Portátil x86-64, 16 GB RAM, SSD 512 GB, Ubuntu 22.04 LTS, conexión simétrica de 300 Mbps; virtualización Docker Desktop.
- **Repositorios y control de versiones:** Organización privada en GitHub; **branch protection** y **semantic-versioning**.

2 Antecedentes y estado del arte

2.1 Evolución de los medios de pago hacia SEPA

El ecosistema europeo de pagos ha experimentado una profunda transformación en las últimas décadas, pasando de sistemas nacionales heterogéneos a un marco unificado bajo la iniciativa **SEPA**. Antes de SEPA, cada país operaba infraestructuras y normas propias para transferencias bancarias y adeudos, lo que complicaba los pagos transfronterizos dentro de Europa. Con la introducción del euro y el objetivo de un mercado único, surgió la necesidad de armonizar los instrumentos de pago. El Consejo Europeo impulsó la creación de SEPA a través del Reglamento (UE) 260/2012, que fijó la migración obligatoria a los nuevos esquemas paneuropeos de transferencia y adeudo en fechas límite (febrero de 2014 para la zona euro).

Así, en 2008 se lanzó el esquema **SEPA Credit Transfer (SCT)** para transferencias de crédito en euros, y en 2009 el **SEPA Direct Debit (SDD)** para adeudos domiciliados. Estos esquemas sustituyeron progresivamente a los medios nacionales, unificando formatos (por ejemplo, el uso obligatorio de *IBAN*) y reglas de funcionamiento en todos los países SEPA. Posteriormente, para atender las demandas de inmediatez, la transferencia instantánea **SCT Inst** (*SEPA Instant Credit Transfer*) entró en funcionamiento en 2017, permitiendo abonar al beneficiario en menos de 10 s. La implantación de SCT Inst ha sido voluntaria hasta ahora, aunque recientemente la UE ha aprobado su obligatoriedad progresiva en 2025 para acelerar su adopción.

En la actualidad, los esquemas SEPA (transferencias estándar e inmediatas, adeudos básicos y B2B) concentran la mayoría de pagos bancarios en euros dentro de Europa. Este salto hacia la unificación de pagos fue liderado por la propia industria bancaria europea. En 2002 los bancos constituyeron el **European Payments Council (EPC)**, órgano de autorregulación que diseña y gestiona los esquemas SEPA. El EPC publicó las primeras *rulebooks* de SCT y SDD en 2008–2009, estableciendo estándares comunes de mensaje (*ISO 20022*¹⁰) y calendarios de liquidación. Cabe destacar que el EPC no es un organismo legislativo de la UE ni un regulador, sino una asociación del sector bancario que actúa de facto como ente normalizador: especifica las reglas de los esquemas utilizados por los **PSP** (*Payment Service Providers* o proveedores de servicios de pago) y coopera con los bancos centrales para operar las infraestructuras de compensación. Gracias a esta colaboración público-privada, a partir de 2014 se completó con éxito la migración de millones de pagos nacionales al formato SEPA, eliminando diferencias entre pagos domésticos y transfronterizos en euros.

2.2 El papel del EPC en la estandarización y el surgimiento de Request-to-Pay

El **European Payments Council (EPC)** ha desempeñado un rol central en la estandarización de los instrumentos de pago SEPA. Tras la implementación de SCT y SDD, el EPC continuó explorando mejoras para la era digital, en línea con las iniciativas del Eurosistema para fomentar pagos electrónicos paneuropeos más eficientes. En noviembre de 2018, el *Euro Retail Payments Board (ERPB)*¹¹ —órgano del BCE que orienta la estrategia de pagos minoristas— lanzó un

¹⁰ISO 20022: estándar internacional para el intercambio de mensajes financieros, basado en XML, empleado en los esquemas SEPA para definir las estructuras de datos de transferencias, adeudos, etc.

¹¹ERPB: foro de alto nivel presidido por el BCE que reúne a autoridades y sector financiero para impulsar la integración y modernización de los pagos minoristas en Europa.

llamado a la acción para desarrollar el concepto de *Request to Pay* (R2P) como nuevo servicio en la zona SEPA.

Atendiendo esta petición, el EPC creó un grupo de trabajo y comenzó a diseñar un esquema formal de Request to Pay durante 2019–2020. Tras una consulta pública, en noviembre de 2020 se publicó el primer *SRTP Scheme Rulebook* (versión 1.0) y se abrió el registro de participantes. El esquema **SEPA Request-to-Pay (SRTP)** entró en vigor el *15 de junio de 2021*, marcando un hito en la evolución de SEPA más allá de los instrumentos tradicionales. Al igual que SCT Inst, la adhesión al esquema RTP es voluntaria; sin embargo, su desarrollo cuenta con fuerte apoyo institucional al considerarse un potenciador de los pagos instantáneos y digitales en Europa. El EPC continúa gestionando y actualizando el esquema (versión 4.0 en 2023), con la expectativa de que Request-to-Pay se integre gradualmente como componente clave del panorama de pagos europeos.

2.3 Funcionamiento técnico de SEPA Request-to-Pay (SRTP)

Request-to-Pay (RTP) es un servicio de mensajería financiera que actúa como capa de solicitud previa al pago. A diferencia de los instrumentos tradicionales (transferencias o adeudos) que mueven fondos, RTP no mueve dinero por sí mismo: permite a un beneficiario (*Payee*) enviar electrónicamente una solicitud de pago a un pagador (*Payer*), quien puede aceptarla o rechazarla antes de iniciarse la transacción monetaria. El servicio funciona 24×7 y añade al flujo de pago un intercambio estructurado de datos (importe, concepto, vencimiento, identidad de las partes, etc.) previo al envío de fondos.

Los mensajes SRTP viajan en tiempo real formateados según ISO 20022¹², lo que facilita su integración con las plataformas SEPA existentes.

Modelo de cuatro esquinas. SRTP adopta la clásica arquitectura *4-corner model*:

¹²Las mensajes SRTP siguen las definiciones ISO 20022 específicas del esquema, permitiendo interoperabilidad con mensajes de pago **pacs/pain** de SCT/SCT Inst.

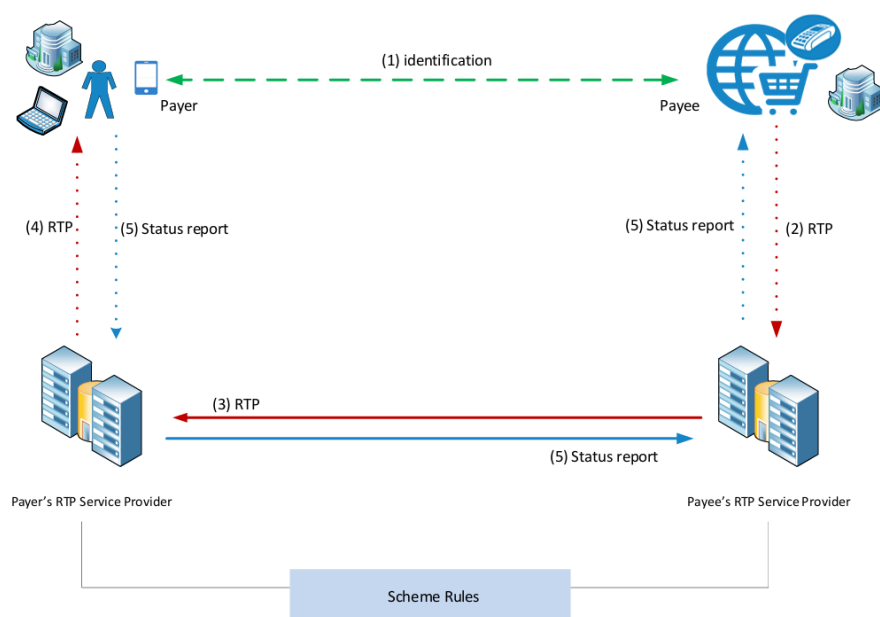


Figura 1: 4CornerModel

Cuadro 2: Pasos del flujo del esquema SEPA Request-to-Pay (SRTP) y su descripción

Paso	Descripción
1. Identificación	Una primera interacción que establece la comunicación entre pagador y beneficiario.
2. Envío de la SRTP al PSP del beneficiario	El beneficiario envía la solicitud de pago SRTP a su PSP, incluyendo todos los datos esenciales del esquema (importe, concepto, vencimiento...).
3. Transmisión al PSP del pagador	El PSP del beneficiario reenvía la SRTP al PSP del pagador.
4. Presentación al pagador	La solicitud se muestra al pagador en el canal acordado (app móvil, web, etc.), permitiendo que revise los detalles.
5. Informe de estado	El resultado (aceptación o rechazo) se comunica de vuelta al beneficiario mediante los PSP correspondientes.

La Operational Scheme Manager (OSM) es el elemento central que mantiene vivo y accesible a todo el ecosistema SRTP. Es un gran directorio seguro y siempre actualizado, gestionado por el EPC, donde quedan registrados todos los PSP y demás actores autorizados: sus identificadores,

los certificados TLS que usan para cifrar las comunicaciones, las claves de firma y las URLs de sus endpoints (puntos finales de API donde se reciben y se envían RTP). Gracias a la OSM, cada PSP puede, en cualquier momento, localizar de forma sencilla y confiable a otro PSP: comprueba automáticamente que el receptor está homologado, que su endpoint está operativo, y que la conexión será segura antes de intercambiar solicitudes o respuestas de pago.

Su misión va más allá de un simple directorio estático: la OSM supervisa y valida de forma continua la disponibilidad y el correcto funcionamiento de todos los endpoints adheridos, publica esta información en el EPC Directory Service (EDS) y notifica de inmediato cualquier incidencia. De este modo, se garantiza que las transacciones RTP fluyan sin interrupciones, con altos estándares de seguridad y cumplimiento de ISO 20022, y que todos los participantes puedan interoperar con total confianza.

En este TFG no se va a simular ni modelar este registro previo en la OSM, asumiremos que todos los actores se encuentran correctamente dados de alta en el directorio. Así, podremos centrarnos exclusivamente en las dinámicas de la solicitud, aceptación, rechazo y cancelación de pagos, dando por hecho que el acceso a los endpoints y la homologación técnica ya están resueltos.

El flujo básico de un intercambio RTP es el siguiente:

Cuadro 3: Pasos del flujo del esquema SRTP y su descripción

Paso/Función	Descripción
1 Crear y enviar RTP	El beneficiario crea el "SEPA Request to Pay" SRTP en el formato normalizado (en un formato acordado bilateralmente con su proveedor). Contiene todos los elementos obligatorios y elementos opcionales que puedan ajustarse al flujo en función de las condiciones comerciales. El beneficiario lo envía al proveedor de servicios SRTP del beneficiario.
2 Validar RTP	El proveedor de servicios SRTP del beneficiario realiza una primera validación del SRTP. Esto incluye, por ejemplo, validación técnica, de seguridad y de formato (por ejemplo, comprobación del IBAN).
2B Rechazo	Si la validación en el paso 2 no tiene éxito, el proveedor de servicios SRTP del beneficiario notifica al beneficiario el rechazo del SRTP, crea un informe de estado negativo y lo envía al beneficiario en el formato acordado con este.
3 Completar y enviar RTP	En caso de validación correcta en el paso 2, el proveedor de servicios SRTP del beneficiario enriquece el SRTP con los elementos necesarios para el enrutamiento en el espacio entre proveedores de servicios SRTP y añade un sello de tiempo.
4 Enrutar RTP	El SRTP se envía al proveedor de servicios SRTP del pagador en función de los mecanismos de enrutamiento establecidos por el PSP.
5 Validar RTP	El proveedor de servicios SRTP del pagador valida el SRTP, incluye la comprobación del identificador del pagador. Esto puede incluir la validación específica del pagador (por ejemplo, si el pagador ha optado por no participar en el servicio, el SRTP es rechazado por defecto).
5B Rechazo	Si la validación en el paso 5 no tiene éxito, el proveedor de servicios SRTP del pagador rechaza el SRTP. El proveedor de servicios SRTP del beneficiario y el beneficiario son informados de este rechazo mediante un código de motivo de no aceptación del RTP.
5C Confirmación positiva funcional	Después de una validación externa en el paso 5, el proveedor de servicios SRTP confirma al pagador que el proveedor de servicios SRTP ha completado con éxito el procedimiento benefical. Esta confirmación es obligatoria solo en el caso de que el beneficiario o el proveedor de servicios SRTP no haya confirmado previamente la positividad funcional.
6 Enviar RTP	En el caso de validación correcta en el paso 5, el proveedor de servicios SRTP envía el documento SRTP al pagador en el formato acordado (el SRTP puede ser convertido en este paso).
7 Evaluar RTP	El pagador decide si aceptar o rechazar el SRTP, determinando el próximo curso de acción.
<i>(continúa en la siguiente página)</i>	

Paso/Función	Descripción
7B Positivo	Si el pagador decide aceptar el SRTP, se envía una respuesta positiva al proveedor de servicios SRTP por parte del pagador.
7C Negativo	Si el pagador rechaza el SRTP, se envía una respuesta negativa al proveedor de servicios SRTP por parte del pagador.
8 Crear/modificar y mandar informe RTP	El proveedor de servicios SRTP crea un informe informativo basado en la decisión del pagador. Si la decisión es negativa (7B/7C), el informe se envía de vuelta al pagador para su revisión. Si el SRTP ya ha sido aceptado o rechazado, surge un caso excepcional donde no se espera un acuse de recibo. El informe debe considerar la fecha de expiración del SRTP. En este caso, el proveedor de servicios SRTP es responsable de notificar al beneficiario la decisión del proveedor de servicios SRTP, incluyendo el código correspondiente. Como resultado, el beneficiario debe representar el SRTP o utilizar otro canal.
9 Enviar informe de estado	El proveedor de servicios SRTP envía un informe de estado actualizado (positivo o negativo) al pagador a través del mismo canal utilizado para el SRTP original, utilizando mecanismos establecidos para la actualización.
10 Proceder y remitir informe	El proveedor de servicios SRTP del beneficiario procesa el informe de estado recibido (positivo o negativo), informa al beneficiario y decide los pasos siguientes previo acuerdo con el beneficiario.
11 Procesar informe de estado	El beneficiario ejecuta las acciones finales tras la recepción del informe de estado: actualización del estado final del registro SRTP, preparación del pago SRTP conciliación, etc.
12 Generar solicitud de actualización de estado	El beneficiario y el proveedor de servicios SRTP del beneficiario pueden enviar una Solicitud de Actualización de Estado si no se ha recibido respuesta hasta la Fecha/Hora de Expiración.
13 Enrutar la solicitud de actualización de estado	La solicitud de actualización de estado al proveedor de servicios SRTP del pagador se enruta a través de la misma vía utilizada para el SRTP original basándose en los mecanismos de enrutamiento establecidos.
14 Validar solicitud de actualización de estado	Tras la recepción de la solicitud de actualización de estado, el Proveedor de Servicios SRTP del pagador comprueba la validez de la Solicitud.
14B Respuesta a la solicitud de actualización de estado	El Proveedor de Servicios SRTP del pagador responde al proveedor de servicios SRTP del beneficiario y, si procede (a través del proveedor de servicios SRTP del beneficiario), al beneficiario (por ejemplo, respuesta del SRTP original no recibida, el pagador aún no ha respondido, etc.).
<i>(continúa en la siguiente página)</i>	

Paso/Función	Descripción
15 Enrutar la solicitud de actualización de estado	En caso de que el pagador aún no haya respondido al SRTP inicial, el proveedor de servicios SRTP del pagador puede enviar la solicitud de actualización de estado al Pagador.
<i>Fin de la tabla</i>	

Hay algunos detalles acerca del esquema que conviene aclarar:

1. Rechazo de un SRTP o una Solicitud de Cancelación (Reject)

Un Reject”se produce cuando un SRTP o una Solicitud de Cancelación no es aceptada antes de ser enviada al siguiente participante en la cadena de pago. El mensaje de rechazo sigue la misma ruta que el SRTP original sin modificar ningún dato, y se incluye un registro con los detalles necesarios para asegurar un rastro de auditoría. Además, el mensaje de rechazo lleva un código de motivo que explica la razón del rechazo. La identificación del SRTP original se hace mediante la referencia única incluida por el proveedor de servicios del receptor. Los rechazos se envían de manera instantánea por el proveedor de servicios SRTP que no puede procesar la solicitud, y estos rechazos pueden ser generados automáticamente en función de comprobaciones técnicas o comerciales, sin intervención del pagador.

2. Respuestas a un SRTP (positiva o negativa)

Cuando un pagador responde a un SRTP, puede aceptar (respuesta positiva) o rechazarlo (respuesta negativa). En ambos casos, el mensaje sigue la misma ruta que el SRTP original, sin alterar los datos, y se envía instantáneamente entre los proveedores de servicios SRTP. Las respuestas negativas incluyen un código de motivo que especifica la razón del rechazo, mientras que las respuestas positivas simplemente confirman la aceptación de la solicitud. Como en el caso de los rechazos, se mantiene un registro detallado de los datos relevantes para asegurar la trazabilidad del proceso y la transparencia en la comunicación entre los proveedores de servicios.

3. Solicitud de Cancelación del SRTP

Una “Request for cancelation” RfC puede ser iniciada por el y se transmite al pagador a través de los proveedores de servicios SRTP. La solicitud sigue la misma ruta que el SRTP original, sin modificar los datos, y debe incluir un código de motivo (atributo AT-R106) que justifique la cancelación. Esta solicitud puede realizarse hasta la fecha de expiración del SRTP, a menos que ya haya sido rechazado, cancelado o expirado. El proveedor de servicios SRTP del pagador verifica la validez de la solicitud antes de reenviarla, y si no se puede procesar, envía una respuesta negativa. Si la cancelación se ejecuta correctamente, el proveedor del pagador envía una respuesta positiva de manera instantánea, manteniendo siempre un registro para garantizar la trazabilidad del proceso.

Casos de uso representativos. El **SEPA Request-to-Pay** se concibió como un servicio versátil, capaz de cubrir desde pagos cotidianos de bajo valor hasta cobros empresariales complejos. No obstante, el caso de uso considerado más transformador—y sobre el que se centra este TFG—es la *sustitución del adeudo directo SEPA (SDD) en pagos recurrentes*. Aun así, existen otros escenarios relevantes que merece la pena describir para contextualizar el alcance potencial de SRTP.

1. Punto de Venta Físico (POS)

- **Descripción:** Este caso de uso de Request to Pay se emplea en tiendas físicas, donde el payee (el comercio) envía una solicitud de pago al payor (el cliente) utilizando un código QR o una tecnología NFC (Near Field Communication). Al escanear el código QR con su móvil o usar NFC en la terminal de pago, el cliente es redirigido a su aplicación bancaria para autorizar el pago de manera inmediata.
- **Proceso de Identificación:**
 - *Identificación del Payor:* El payor se autentica directamente en su aplicación bancaria, generalmente mediante su número de cuenta bancaria, número de tarjeta, o métodos de autenticación biométrica o PIN, según lo permita la aplicación del banco.
 - *Identificación del Payee:* El payee está identificado en el sistema mediante un ID de comercio asociado a la terminal de pago o al código QR/NFC escaneado por el cliente. Estos elementos proporcionan los datos necesarios para vincular la solicitud de pago al comercio correspondiente.
- **Proceso de Pago:**
 - El cliente escanea el código QR o se conecta mediante NFC a la terminal de pago.
 - La aplicación bancaria del cliente recibe la solicitud de pago con el monto y la referencia.
 - El cliente revisa la información y autoriza el pago, un proceso rápido y conveniente, crucial en entornos físicos donde la velocidad es esencial.
- **Diferencias Clave:**
 - La rapidez y conveniencia son fundamentales en este caso de uso. El proceso de identificación y autorización es sencillo y rápido, requiriendo solo la confirmación del cliente a través de su banco, típicamente con autenticación biométrica o PIN.
 - Es ideal para transacciones de bajo valor donde la experiencia del cliente es un factor determinante.

2. Comercio Electrónico (E-commerce)

- **Descripción:** En este caso, el payee (el comercio electrónico) envía una solicitud de pago al payor (el cliente) durante el proceso de checkout o mediante un enlace de pago enviado a través de una aplicación bancaria o correo electrónico. El cliente es redirigido a su aplicación bancaria, donde debe autenticarse y revisar los detalles antes de aprobar el pago.
- **Proceso de Identificación:**
 - *Identificación del Payor:* La identificación implica un proceso de autenticación multifactor (por ejemplo, contraseña, token de seguridad o autenticación biométrica) para garantizar la seguridad de la transacción, realizado en la aplicación bancaria o la página de pago del comercio.
 - *Identificación del Payee:* El payee se identifica mediante un ID de comercio electrónico que incluye su nombre comercial, identificador fiscal, número de cuenta o un identificador único en la plataforma de pagos.
- **Proceso de Pago:**

- El cliente recibe la solicitud de pago a través de un enlace en el checkout o en su aplicación bancaria.
- El cliente se autentica en su aplicación bancaria, revisa el monto, la referencia y los detalles del comerciante, y aprueba el pago.
- Una vez validada, el dinero se transfiere de manera segura.
- **Diferencias Clave:**
 - A diferencia del punto de venta físico, el proceso en comercio electrónico es más lento debido a la autenticación adicional y la revisión en línea.
 - La seguridad es prioritaria, dado el mayor riesgo de fraude en transacciones digitales.

3. Facturación Electrónica (E-invoicing)

- **Descripción:** Común en empresas que envían facturas electrónicas, el payee (la empresa emisora) envía una solicitud de pago con los detalles de la factura al payor (el cliente). El cliente recibe la solicitud en su correo electrónico o aplicación bancaria, pudiendo revisar los detalles antes de decidir cuándo pagar.
- **Proceso de Identificación:**
 - *Identificación del Payor:* El payor se identifica mediante su número de cliente, correo electrónico o número de cuenta bancaria vinculado a la empresa emisora.
 - *Identificación del Payee:* El payee se identifica por su NIF (Número de Identificación Fiscal) o un ID de facturación electrónica único, asegurando la verificación de la entidad receptora.
- **Proceso de Pago:**
 - El cliente recibe la factura y la solicitud de pago por correo o notificación bancaria.
 - Tras validar la información, autoriza el pago a través de su aplicación bancaria, completando la transacción.
- **Diferencias Clave:**
 - Ofrece flexibilidad, permitiendo al cliente revisar la factura antes de pagar.
 - La identificación del payee incluye detalles fiscales, incrementando la seguridad en la validación.

4. Pagos Recurrentes

- **Descripción:** Ideal para suscripciones o pagos periódicos (streaming, software en la nube, gimnasios), el payor autoriza la primera transacción para suscribirse. Los pagos posteriores se gestionan automáticamente según la periodicidad acordada, sin intervención adicional.
- **Proceso de Identificación:**
 - *Identificación del Payor:* El payor autoriza la primera transacción con autenticación en su aplicación bancaria (cuenta, contraseña o biométrica).
 - *Identificación del Payee:* El payee se identifica por su ID de suscripción y número de cuenta para configurar los pagos recurrentes.
- **Proceso de Pago:**

- El cliente autoriza el pago inicial.
- Los cobros posteriores se realizan automáticamente según la periodicidad (mensual, anual, etc.).
- **Diferencias Clave:**
 - Se distingue por la automatización tras la autorización inicial, reduciendo la intervención del cliente.
 - La comodidad y la automatización son sus principales ventajas.

5. Pagos de Grandes Montos

- **Descripción:** Utilizado en transacciones de alto valor (compras importantes, bienes de gran valor), este caso requiere métodos adicionales de autenticación para garantizar la seguridad.
- **Proceso de Identificación:**
 - *Identificación del Payor:* El payor usa autenticación robusta (OTP, tokens de seguridad o multifactor como PIN y biométrica) para validar la transacción.
 - *Identificación del Payee:* El payee es una entidad registrada con un identificador único (ID de comerciante), asegurando el destinatario correcto.
- **Proceso de Pago:**
 - El payor revisa el monto y autoriza el pago con autenticación adicional.
 - Tras verificar la identidad, el pago se completa.
- **Diferencias Clave:**
 - Requiere autenticación más fuerte debido al alto riesgo de fraude en grandes sumas.
 - La seguridad es el factor más crítico, priorizando la protección contra fraudes y errores.

2.4 RTP dentro del ecosistema de pagos: arquitectura en capas

El ecosistema de pagos SEPA se puede describir mediante capas jerárquicas, desde los usuarios finales hasta las infraestructuras financieras que ejecutan las transacciones. A continuación se detallan las principales capas del modelo actual de pagos SEPA:

Capa de usuarios finales En la capa más alta se encuentran el *pagador* y el *beneficiario*. Son quienes inician y reciben los pagos, respectivamente, ya sea personas o empresas involucradas en una transacción.

Capa de iniciación o interacción Es donde se produce la solicitud o autorización del pago a través de algún canal o interfaz. Por ejemplo, en pagos SEPA tradicionales el pagador suele autorizar una transferencia mediante la banca online o una aplicación móvil. En entornos de comercio electrónico o físico, esta capa correspondería al proceso de *checkout* (por ejemplo, introduciendo datos en una pasarela de pago online o mediante un TPV/POS en tienda).

Capa de proveedores de servicios de pago (PSP) Aquí actúan las entidades financieras o PSP de cada parte. Típicamente, el banco del pagador y el banco del beneficiario son quienes ofrecen las cuentas bancarias y servicios de pago a sus clientes. Estas entidades facilitan la emisión y recepción de órdenes de pago en nombre de los usuarios finales.

Capa de esquemas de pago Es el conjunto de reglas y estándares comunes que permiten la interoperabilidad entre todos los PSP. En SEPA, por ejemplo, existen los esquemas de transferencia crediticia (SCT, SCT Inst) y adeudo directo (SDD), definidos por el European Payments Council (EPC). Cada esquema asegura que, independientemente del banco involucrado, un pago en euros siga las mismas normas y formatos. Un esquema de pago no es el movimiento del dinero en sí, sino el conjunto de reglas, mensajes e infraestructura técnica que orquesta cómo se inician y gestionan esos pagos.

Capa de infraestructuras de compensación Son las plataformas interbancarias que procesan las transacciones según las reglas del esquema. Estas infraestructuras se encargan de intercambiar las órdenes de pago entre el banco emisor y el banco receptor, aplicando compensación multilateral si procede.

Capa de liquidación Es la capa más baja, donde se realiza la liquidación final de fondos entre bancos. Normalmente ocurre en los bancos centrales u organismos de compensación: por ejemplo, las obligaciones netas resultantes de muchas transacciones SEPA pueden liquidarse en TARGET2 (sistema del Banco Central Europeo). En pagos inmediatos, la liquidación suele ser casi en tiempo real (por ejemplo, mediante TIPS que liquida cada operación al momento en cuentas del banco central).



Figura 2: Layer

Esta arquitectura por capas garantiza que un pago iniciado por un usuario en un banco pueda llegar a otro usuario en distinto banco de forma segura, eficiente e interoperable en toda la zona euro. Cada capa agrega funciones específicas: los usuarios generan órdenes, los PSP las gestionan, los esquemas proporcionan las reglas comunes, y las infraestructuras las ejecutan y asientan los fondos en última instancia.

RTP es un nuevo servicio/esquema incorporado en la arquitectura SEPA que se sitúa principalmente en la capa de esquema de pago, actuando como una capa adicional de mensajería sobre los instrumentos de pago existentes.

Ejemplo comparado: pago con tarjeta vs. RTP

Como muestra el esquema comparativo, los pagos con tarjeta (ej. Visa/Mastercard) históricamente han operado con una arquitectura de funciones similar a la de SEPA, pero con diferencias en los actores y procesos de cada capa:

Usuarios (pagador/beneficiario) En tarjetas, el pagador es el *titular de la tarjeta* y el beneficiario es el *comercio* que recibe el pago. En esencia es equivalente al ordenante y beneficiario

de una transferencia, con la diferencia de que el pagador utiliza un instrumento distinto (su tarjeta en lugar de una cuenta bancaria directa).

PSP / entidades En el modelo de cuatro partes de las tarjetas interviene el *banco emisor* (emite la tarjeta al pagador) y el *banco adquirente* (procesa pagos para el comerciante). Estos roles son análogos a la entidad del pagador y del beneficiario en SEPA, pero en el mundo tarjeta suelen implicar acuerdos específicos (p. ej. el comerciante contrata un adquirente para aceptar Visa/Mastercard). En cambio, en SEPA cualquier banco puede enviar o recibir transferencias para un cliente sin acuerdos individuales con cada comercio, ya que todos siguen el esquema común.

Esquema de pago Las tarjetas operan bajo esquemas propietarios como Visa, Mastercard, etc., que definen reglas, formatos de mensajes (p. ej. mensajes de autorización y liquidación) y que actúan también como redes de procesamiento. Son equivalentes a los esquemas SEPA en cuanto a que proveen interoperabilidad, pero controlados por empresas particulares. El esquema Visa/Mastercard indica cómo se autoriza una compra, cómo se liquida posteriormente y fija también aspectos comerciales como las tasas de intercambio entre emisor y adquirente. En SEPA, el esquema SRTP + SCT Inst provee una funcionalidad comparable de solicitud y pago, pero dentro de un marco colaborativo paneuropeo. De hecho, SRTP adopta un modelo muy similar al de tarjetas de cuatro partes (pagador, beneficiario y sus respectivos proveedores), tanto que el EPC prevé que incluso podrían llegar a existir comisiones de intercambio análogas en este ecosistema¹³ (aunque inicialmente SRTP nace sin tarifas de intercambio explícitas).

Infraestructura de compensación En los pagos con tarjeta, la red del esquema (p. ej. VisaNet) se encarga de la autorización instantánea de la transacción y de la compensación/clearing de las transacciones entre emisores y adquirentes. Visa o Mastercard centralizan el intercambio de mensajes financieros. En SEPA, por el contrario, las compensaciones suelen realizarse a través de múltiples infraestructuras (por ejemplo, cámaras como STEP2 o servicios inmediatos como TIPS) que no pertenecen a una sola empresa sino que son parte del ecosistema colaborativo europeo. Con Request to Pay, la mensajería de solicitud viaja por la red designada (p. ej. la plataforma R2P de EBA Clearing) y el pago resultante se compensa a través de los *rails* SEPA existentes (p. ej. RT1/TIPS para instantáneas).

Liquidación final En ambos casos, finalmente hay un traspaso de fondos entre bancos. En las tarjetas, las marcas de tarjeta calculan las obligaciones netas entre cada banco emisor y adquirente y típicamente las liquidan al final del día a través de cuentas en un banco central u otros mecanismos interbancarios. En SEPA, cada transferencia individual (especialmente si es instantánea) puede liquidarse inmediatamente en el banco central. Desde el punto de vista de capas, ambos mundos terminan convergiendo en la necesidad de que el dinero se ajuste entre las cuentas de los bancos participantes.

¹³<https://redbridgedta.com>

SEPA (Request-to-Pay + SCT Inst) Tarjetas (Visa/Mastercard)

Usuarios finales: - Pagador (ordenante) - Beneficiario (cobrador)	Usuarios finales: - Titular de la tarjeta - Comercio (beneficiario)
Interacción / Iniciación: - Solicitud de pago (RTP) - App/web (banca, e-com)	Interacción / Iniciación: - Presentación tarjeta / POS - Pasarela de pago online
PSP / Entidades: - Banco del pagador - Banco del beneficiario	PSP / Entidades: - Banco emisor (tarjeta) - Banco adquirente
Esquema de pago: - SRTP (mensajería) - + Transferencia SEPA Inst (SCT Inst)	Esquema de pago: - Esquema de tarjeta (Visa/MC) - (autorización & cobro)
Infraestructura compensación: - Red interbancaria SEPA (p.ej. RT1/TIPS)	Infraestructura compensación: - Red del esquema de tarjeta
Infraestructura liquidación: - Liquidación en banco central (TARGET2/TIPS)	Infraestructura liquidación: - Liquidación interbancaria (p.ej. BC)

Figura 3: LayerComp

¿Qué simplifica o elimina Request to Pay respecto al modelo de tarjetas?

Principalmente, elimina intermediarios dedicados y procesos redundantes. Por ejemplo, en un pago SRTP + SCT Inst no es necesario un procesador/acquirente específico ni una red de tarjetas propietaria, ya que los propios bancos de pagador y beneficiario se comunican directamente mediante el esquema común¹⁴. Esto puede reducir costes de aceptación para el comercio (evitando comisiones elevadas de tarjetas) y simplifica la integración: el comercio recibe el dinero directamente en su cuenta bancaria vía SEPA, sin pasos intermedios de recibir fondos a través de entidades de tarjeta y luego liquidarlos. Además, no se requiere que el pagador proporcione datos sensibles como el PAN de tarjeta o incluso su IBAN al comercio; la solicitud llega por canales bancarios seguros y el cliente simplemente autoriza en su entorno bancario¹⁵. En resumen, Request to Pay se apoya

¹⁴<https://cpg.de>, <https://redbridgedta.com>

¹⁵<https://docs.monei.com>

en la infraestructura bancaria existente (cuentas y pagos inmediatos) para ofrecer una experiencia similar a la de tarjeta, pero con menos capas propietarias, aprovechando la red SEPA ya desplegada en toda Europa.

3 Diseño e Implementación

Para implementar un sistema completo de *Request To Pay*, se ha desarrollado un servidor HTTP que expone una API RESTful y un cliente web en tiempo real. El objetivo de este apartado es detallar cómo se ha llevado a cabo esta implementación, así como las fases de desarrollo involucradas.

3.1 Fundamentos Teóricos

Antes de profundizar en la implementación, es fundamental aclarar los dos pilares teóricos en los que se basa el sistema:

- **Servidor HTTP:** Una aplicación que permanece a la escucha en un puerto de red, esperando conexiones TCP de clientes capaces de comunicarse mediante el protocolo HTTP. Una vez establecida la conexión, el servidor recibe un mensaje estructurado con los siguientes componentes:
 - *Línea de petición:* Indica el método (GET, POST, etc.), la ruta del recurso solicitado y la versión del protocolo.
 - *Cabeceras:* Aportan metadatos, como el tipo de contenido, credenciales o longitud del mensaje.
 - *Cuerpo* (opcional): Contiene datos adicionales, si el método lo requiere.

El servidor interpreta la ruta, determina qué componente interno debe procesarla, ejecuta la lógica correspondiente y genera una respuesta formada por:

- *Línea de estado:* Incluye un código de resultado (por ejemplo, 200 OK, 404 Not Found, 500 Internal Server Error).
- *Cabeceras:* Describen la respuesta.
- *Cuerpo* (opcional): Contiene datos, como HTML o JSON.

Tras enviar la respuesta, la conexión puede cerrarse o mantenerse activa para futuras peticiones, dependiendo de la versión del protocolo y las cabeceras de control. En esencia, el servidor HTTP funciona como el centro de operaciones que recibe todas las solicitudes y coordina el acceso a la lógica y los datos de la aplicación.

- **API RESTful:** Se basa en los principios de la arquitectura REST (*Representational State Transfer*¹⁶) aplicados al protocolo HTTP para exponer recursos de forma uniforme y predecible. Sus características principales son:
 - Cada entidad del dominio (por ejemplo, un usuario o una petición RTP) se representa mediante una URL estable.
 - Los verbos HTTP (POST, GET, PUT, DELETE) describen operaciones como creación, consulta, modificación o eliminación de recursos.
 - El servidor es *sin estado*, por lo que cada solicitud contiene toda la información necesaria, facilitando la escalabilidad horizontal.¹⁷

¹⁶Transferencia de Estado Representacional

¹⁷Es decir, permite añadir o retirar servidores sin necesidad de compartir sesiones en memoria.

- La uniformidad de los códigos de estado y los formatos de representación asegura que clientes heterogéneos consuman la API de manera predecible.
- Herramientas como cachés, control de versiones en URLs o cabeceras, y negociación de contenido permiten evolucionar la interfaz sin afectar a los consumidores existentes.

En conjunto, el servidor HTTP actúa como el camino por donde viajan las solicitudes, mientras que la API RESTful establece las reglas claras y fáciles de mantener para que ese camino conecte eficientemente la lógica del servidor con los diversos clientes que dependen de ella.

3.2 Tecnologías utilizadas

El desarrollo del proyecto se ha llevado a cabo en un entorno virtual Python que aísla las dependencias y permite reproducir la instalación mediante el comando `pip install -r requirements.txt`. Este enfoque asegura consistencia y portabilidad, facilitando tanto la colaboración entre desarrolladores como el despliegue en entornos diversos.

Backend. El núcleo del backend se fundamenta en Python 3 y Flask, un micro-framework que proporciona un servidor WSGI, un sistema de enrutado eficiente y una integración fluida con el estándar HTTP. Sobre esta base, se incorpora Flask-socketIO, un middleware que habilita la negociación de WebSockets, permitiendo servir tráfico HTTP y comunicación bidireccional en el mismo puerto de manera transparente. Para la gestión de datos, se emplea SQLAlchemy a través de Flask-SQLAlchemy, lo que facilita la representación de modelos de dominio como objetos Python, mientras que SQLite actúa como una base de datos ligera y autónoma durante el desarrollo, eliminando la necesidad de un servidor externo para transacciones básicas. Este núcleo se complementa con utilidades de la librería estándar de Python, incluyendo:

- `hashlib`, para firmar transacciones de estado mediante el algoritmo SHA-256;
- `datetime`, para registrar marcas temporales en los logs.

Frontend. En el lado del cliente, se ha implementado un frontend estático basado en HTML5, CSS3 y JavaScript ES6, priorizando la ligereza al evitar frameworks complejos. La maquetación adaptativa se logra mediante Bootstrap 5, cargado vía CDN, y la iconografía se enriquece con Font Awesome, también distribuido por CDN. La comunicación en tiempo real se establece con el cliente Socket.IO 4.x, que conecta vía WebSocket al mismo host y puerto que Flask, mientras que las peticiones REST se realizan de forma nativa con la Fetch API, sin depender de librerías adicionales.

Herramientas de desarrollo. El entorno de trabajo se ha centrado en Visual Studio Code como editor principal, aprovechando sus extensiones para optimizar la gestión del proyecto y el desarrollo del código. El control de versiones se ha gestionado con Git, utilizando ramas específicas para cada funcionalidad nueva, lo que asegura un desarrollo ordenado y trazable. Para las pruebas manuales de la API, se ha empleado Postman, donde se diseñó una colección de peticiones parametrizadas que serán detalladas en secciones posteriores del documento.

El uso de un entorno virtual junto con dependencias consolidadas establece una base robusta para el despliegue de la aplicación en entornos más exigentes, como contenedores, nubes públicas o servidores locales, preservando la integridad de su arquitectura fundamental.

3.3 Estructura y funcionamiento

El código del proyecto se estructura en dos componentes principales, **backend** y **frontend**, interconectados mediante los protocolos HTTP y WebSockets. Esta división, diseñada de manera intencionada, responde a la necesidad de lograr un desarrollo ordenado, eficiente y preparado para futuros crecimientos, considerando que trabajé en el proyecto de forma individual. Al separar el *backend*, encargado de la lógica de negocio, el manejo de datos y la comunicación con el cliente, del *frontend*, centrado en la interfaz de usuario y la experiencia interactiva, se obtiene una arquitectura clara y modular. Esta organización aporta múltiples ventajas: mejora la mantenibilidad al permitir identificar y corregir errores de manera localizada, simplifica la incorporación de nuevas funcionalidades sin alterar otras partes del sistema y refleja fielmente la arquitectura empleada en entornos de producción reales, lo que facilita una transición fluida hacia despliegues en contenedores, nubes públicas o servidores locales. Además, esta separación promueve la reutilización de código, ya que el *backend* puede servir a múltiples clientes (como aplicaciones móviles o de escritorio) y el *frontend* puede adaptarse a diferentes dispositivos sin modificar la lógica subyacente. A continuación, se describen los archivos que componen cada una de estas partes, su rol específico en el proyecto y el fragmento de código más relevante.

```
(venv) → SEPA_RTP git:(main) X tree -L 1 backend frontend
backend
├── __pycache__
├── app.py
├── config.py
├── ext_socketio.py
├── models.py
├── requirements.txt
├── routes.py
├── routes_actors.py
├── rtp.db
├── services.py
├── utils.py
└── utils_roles.py
frontend
├── RTP.html
├── RTP.js
├── app.js
├── index.html
└── styles.css
```

Figura 4: Tree

3.3.1 Backend

- **app.py** es el punto de arranque, para lanzar la aplicación debo ejecutar dentro de mi venv "python app.py". En este fichero se crea la instancia de Flask, aplica la configuración, inicializa la base de datos y arranca Socket.IO. Al mismo tiempo registra el blueprint con todos los endpoints y expone una ruta raíz que sirve directamente index.html, de manera que el mismo proceso pueda funcionar como servidor de API y de archivos estáticos.

```
(venv) → backend git:(main) X python app.py
* Restarting with stat
* Debugger is active!
* Debugger PIN: 873-234-129
(37109) wsgi starting up on http://127.0.0.1:5000
```

Figura 5: Arranque

Tras ejecutarlo basta con poner "http://127.0.0.1:5000" en el buscador para conectarnos al servidor.

```
1     STATIC_FOLDER = os.path.join(os.path.dirname(__file__),
2     ' ../frontend')
3     app = Flask(__name__, static_folder=STATIC_FOLDER)
4     app.config.from_object(Config)
5     db.init_app(app)
6     # Servir index.html en la raíz
7     @app.route('/')
8     def home():
9         return app.send_static_file('index.html')
10
11     # Servir archivos estáticos desde el directorio frontend
12     @app.route('/<path:filename>')
13     def serve_static(filename):
14         return send_from_directory(app.static_folder, filename)
15
16     # Registrar el blueprint con todos los endpoints de RTP
17     app.register_blueprint(rtp_blueprint)
18
19     @socketio.on('join')
20     def handle_join(data):
21         actor_id = data['actor_id']
22         actor = Actor.query.get(actor_id)
23         if actor:
24             room = f"{actor.role}_{actor_id}"
25             join_room(room)
26             print(f"Actor {actor_id} se une a la sala {room}")
27
28     if __name__ == '__main__':
29         webbrowser.open("http://127.0.0.1:5000/")
30         socketio.run(app, debug=True)
```

Listing 1: Configuración y arranque del servidor Flask-SocketIO

- **config.py** contiene la configuración: URI de SQLite y parámetro para desactivar el tracker de SQLAlchemy. Facilita cambiar de motor BD en modificaciones futuras solamente cambiando esta clase.

```
1     basedir = os.path.abspath(os.path.dirname(__file__))
2
3     class Config:
4         SQLALCHEMY_DATABASE_URI = 'sqlite:/// ' +
5         os.path.join(basedir, 'rtp.db')
6         SQLALCHEMY_TRACK_MODIFICATIONS = False
```

Listing 2: Configuración SQLAlchemy y SQLite

- **models.py** define el modelo de datos que se va utilizar y contiene un ORM¹⁸ para interactuar con una base de datos relacional. Aquí definimos tres clases principales que representan tablas en la base de datos: Actor, RTP y log. Esto se hace así para mayor facilidad a la hora de realizar operaciones como crear, leer, actualizar y eliminar registros. Cada una de estas clases corresponde con una tabla en la base de datos y cada atributo de cada clase con una columna de su tabla correspondiente. Además se incluye un método todict para serializar los datos a formato JSON para enviarlos a través de una API. A continuación se muestra únicamente una de las 3 clases ya que ambas tienen la misma estructura.

```
1      from flask_sqlalchemy import SQLAlchemy
2
3      db = SQLAlchemy()
4
5      class Actor(db.Model):
6          __tablename__ = 'actor'
7          id = db.Column(db.Integer, primary_key=True)
8          username = db.Column(db.String(50), unique=True)
9          password = db.Column(db.String(50))
10         name = db.Column(db.String(50), nullable=False)
11         role = db.Column(db.String(20), nullable=False)
12
13         photo_url = db.Column(db.String(200), nullable=True)
14         # URL o base64
15         iban = db.Column(db.String(34), nullable=True)
16         balance = db.Column(db.Float, default=0.0)
17
18         # Campo nuevo: psp_id, que referencia a otro Actor que
19         # es PSP
20         psp_id = db.Column(db.Integer,
21                             db.ForeignKey('actor.id'), nullable=True)
22         # Relación para que se pueda acceder con actor.psp
23         psp = db.relationship('Actor', remote_side=[id])
24
25         def to_dict(self):
26             return {
27                 "id": self.id,
28                 "username": self.username,
29                 "name": self.name,
30                 "role": self.role,
31                 "photo_url": self.photo_url,
32                 "iban": self.iban,
33                 "balance": self.balance,
34                 "psp_id": self.psp_id # Podr amos incluir
35                                     m s info del psp, etc.
36             }
```

Listing 3: Modelo ORM + clase Actor

¹⁸Object-Relational Mapping

- **utils.py** agrega métodos auxiliares que son llamados desde otros ficheros. Son métodos de utilidad y se han puesto en este fichero con el objetivo de no sobrecargar los ficheros principales. Algunas de las funcionalidades que incluye son *cambiar_estado_rtp*, *rechazar_rtp* o *validar_iban*.

```

1      def cambiar_estado_rtp(db, rtp_obj, new_status):
2          old_status = rtp_obj.status
3          rtp_obj.status = new_status
4          db.session.commit()
5
6          # Generar un hash
7          hash_input =
8              f"{rtp_obj.id}{rtp_obj.iban}{rtp_obj.amount}{old_status}{new_status}".encode()
9          hash_value = hashlib.sha256(hash_input).hexdigest()
10
11         nuevo_log = Log(
12             rtp_id=rtp_obj.id,
13             old_status=old_status,
14             new_status=new_status,
15             hash_value=hash_value
16         )
17         db.session.add(nuevo_log)
18         db.session.commit()
19
20         return {
21             "message": f"RTP {rtp_obj.id} actualizado de {old_status}
22                 a {new_status}"
23         }
24
25     def rechazar_rtp(db, rtp_obj, motivo):
26         return cambiar_estado_rtp(db, rtp_obj, "rechazado")
27
28     def validar_iban(iban, rtp_obj):
29         iban = rtp_obj.iban
30         if not iban or not isinstance(iban, str) or len(iban) < 15:
31             return cambiar_estado_rtp(db, rtp_obj, "rechazado")
32         if not (iban[:2].isalpha() and iban[2:].replace(' ',
33             '').isalnum()):
34             return cambiar_estado_rtp(db, rtp_obj, "rechazado")
35         return {
36             "message": f"RTP {rtp_obj.id} validado con IBAN correcto"
37         }

```

Listing 4: Funciones utilitarias

- **utils.roles.py** contiene el decorador *rolrequired* que inspecciona el actorid recibido en el cuerpo JSON y bloquea la ruta si el rol no coincide con el requerido. De esta manera se asegura de cada actor tenga acceso a sus propias funciones.

```

1      def role_required(required_role):
2          """
3          Decorador que verifica que el actor que invoca el endpoint
4          tenga el rol adecuado.
5          Se espera que en el body JSON llegue "actor_id".
6          """
7      def decorator(f):
8          @wraps(f)

```



```

9         def wrapper(*args, **kwargs):
10             data = request.get_json() or {}
11             actor_id = data.get('actor_id')
12             if not actor_id:
13                 return jsonify({"error": "Se requiere actor_id para esta acci n"}), 400
14
15             actor = Actor.query.get(actor_id)
16             if not actor:
17                 return jsonify({"error": "Actor no encontrado"}), 404
18
19             if actor.role != required_role:
20                 return jsonify({"error": f"Rol '{actor.role}' no tiene acceso a esta acci n"}), 403
21
22             # Si el rol coincide, continuamos
23             return f(*args, **kwargs)
24         return wrapper
25     return decorator

```

Listing 5: Decorador para control de funciones

- **services.py** contiene la capa de negocio. Cada función del fichero orquesta la consulta/actualización de modelos, invoca las utilidades de estado y emite eventos Socket.IO a la room adecuada. A través de todas estas funciones se lleva a cabo el flujo RTP, que posteriormente explicaré.
- **routes.py** contiene el Blueprint que expone la máquina de estados RTP (cinco rutas POST) y utilidades de autenticación, perfil y logs. Cada endpoint invoca a su homónimo en **services.py** y devuelve el JSON estandarizado. Algunas de estos endpoints que no afectan directamente sobre el flujo RTP son el proceso de login (ya que se trata de simular la página de un banco) o el apartado de perfil donde puedes modificar algunos elementos.

A cada uno de estos endpoints le corresponde su fichero o apartado .html que veremos en posteriores apartados del documento.

```

1         rtp_blueprint = Blueprint('rtp', __name__)

```

Listing 6: Declaración blueprint rtp

```

1         @rtp_blueprint.route('/login', methods=['POST'])
2         def login():
3             data = request.get_json()
4             username = data.get('username')
5             password = data.get('password')
6             if not username or not password:
7                 return jsonify({"error": "Faltan credenciales"}), 400
8
9             actor = Actor.query.filter_by(username=username).first()
10            if not actor:
11                return jsonify({"error": "Usuario no existe"}), 404
12
13            if actor.password != password:
14                return jsonify({"error": "Contrase a incorrecta"}), 401

```

```

15
16         # Login exitoso
17         return jsonify({
18             "message": "Login correcto",
19             "actor_id": actor.id,
20             "role": actor.role,
21             "name": actor.name
22         })

```

Listing 7: Endpoint login

```

1         @rtp_blueprint.route('/profile', methods=['POST'])
2         @role_required('payer')
3         def update_profile():
4             data = request.get_json() or {}
5             actor_id = data.get('actor_id')
6             actor = Actor.query.get(actor_id)
7             if not actor:
8                 return jsonify({"error": "Actor no encontrado"}),
9                     404
10
11             # Campos opcionales
12             new_photo = data.get('photo_url')
13             new_iban = data.get('iban')
14             new_balance = data.get('balance')
15
16             if new_photo is not None:
17                 actor.photo_url = new_photo
18             if new_iban is not None:
19                 actor.iban = new_iban
20             if new_balance is not None:
21                 actor.balance = float(new_balance)
22
23             db.session.commit()
24             return jsonify({"message": "Perfil actualizado",
25                             "actor": actor.to_dict()})

```

Listing 8: Endpoint profile

- **routes actors.py** contiene un pequeño blueprint aparte para el alta de actores, esto se ha hecho así para demostrar cómo escalar la API sin engordar el fichero principal.

```

1         actors_blueprint = Blueprint('actors', __name__)
2
3         @actors_blueprint.route('/actors', methods=['POST'])
4         def create_actor():
5             data = request.get_json()
6             name = data.get('name')
7             role = data.get('role') # 'beneficiary',
8                                     'psp_beneficiary', 'psp_payer', 'payer'
9
10            if not name or not role:
11                return jsonify({"error": "Faltan campos requeridos:
12                    name, role"}), 400
13
14            # Validar que el role sea uno de los cuatro permitidos:

```

```

13         valid_roles = ['beneficiary', 'psp_beneficiary',
14                        'psp_payer', 'payer']
15         if role not in valid_roles:
16             return jsonify({"error": f"Rol '{role}' no v lido"}),
17             400
18
19         actor = Actor(name=name, role=role)
20         db.session.add(actor)
21         db.session.commit()
22
23         return jsonify({"message": "Actor creado", "id": actor.id,
24                        "role": actor.role}), 201

```

Listing 9: Declaración blueprint actores

- **ext socketio.py** simplemente es un fichero de conveniencia que crea la instancia socketIO. Separar la instancia de Socket.IO en un fichero independiente permite mantener una arquitectura limpia y flexible: evita importaciones circulares entre módulo, se integra sin fricción en el patrón application factory de Flask al poder llamarse luego socketio.initapp desde app.py. Así centralizamos en un único punto la configuración o el cambio de motor de concurrencia facilitando la escalabilidad.

```

1         from flask_socketio import SocketIO, join_room, emit
2
3         socketio = SocketIO()

```

Listing 10: Instancia global de socketIO

- **rtp.db** es una base de datos sencilla que se crea sola la primera vez que inicias la aplicación. Cuando arrancas el programa, las herramientas explicadas anteriormente crean este archivo automáticamente. Este archivo guarda toda la información del proyecto, como las tablas y datos, sin necesidad de instalar nada más ni usar un servidor aparte. Esto hace que sea muy fácil empezar a trabajar en el proyecto, ya sea para desarrollarlo, hacer pruebas o mostrar cómo funciona, porque solo necesitas instalar lo básico.

Cuando el proyecto crezca y se necesite manejar más usuarios al mismo tiempo, hacer copias de seguridad mientras está funcionando o tener varias copias de la base de datos, se puede cambiar a una base de datos más potente, como PostgreSQL o MySQL. Esto se hace simplemente cambiando una dirección de conexión, sin modificar casi nada del programa.

3.3.2 Frontend

- **index.html** define el formulario de login inicial, el navbar colapsable y cinco bloques: homeDashboard, seccionCuentas, seccionTarjetas, seccionPerfiles y seccionRTP, de las cuales la única relevante es la seccionRTP, las demás son relleno para simular una página de un banco.

El documento carga Bootstrap 5, Font Awesome, Google Fonts y la hoja styles.css.

Al final del documento se incluye app.js y RTP.js de modo que la misma página sirve toda la experiencia de usuario sin recargas.

- **app.js** desempeña un papel central como el núcleo de la interfaz de usuario, orquestando toda la experiencia de la aplicación de página única (*single-page application*, SPA) del proyecto.

Desde su carga inicial, este fichero establece las bases para una interacción fluida y dinámica, integrando las siguientes funcionalidades clave:

- **Establecimiento de la conexión WebSocket.** Al cargarse, inicia una conexión WebSocket con el servidor mediante una instrucción adecuada, asegurando que los eventos emitidos por el backend lleguen al navegador en tiempo real. Tras un inicio de sesión exitoso, envía un evento `join` que suscribe al usuario a la sala correspondiente, permitiendo recibir actualizaciones instantáneas de cambios de estado.
 - **Gestión del proceso de autenticación.** Intercepta el envío del formulario de inicio de sesión, realiza una solicitud `POST /login`, almacena en memoria el identificador (`currentActorId`) y el rol (`currentActorRole`) devueltos por la API, y desbloquea la navegación por la aplicación, ofreciendo una transición suave al usuario.
 - **Navegación tipo SPA.** Incluye una función que muestra u oculta secciones según la opción seleccionada en la barra de menús, evitando recargas completas de la página y simulando la experiencia de una aplicación nativa. Esta funcionalidad garantiza una navegación ágil y eficiente.
 - **Carga del panel principal.** Al acceder al panel “Inicio”, invoca un cargador de datos que obtiene el perfil del usuario mediante una solicitud `fetch`, formatea el saldo con separadores de miles para mayor claridad, enmascara el IBAN por razones de seguridad y ajusta dinámicamente el tamaño de fuente para que el monto se ajuste perfectamente dentro del círculo de balance.
 - **Edición del perfil.** Ofrece un módulo que precarga el formulario con los datos actuales del usuario, permite su edición y, al enviarlo, actualiza la información en el servidor a través de una solicitud `POST /profile`, mostrando mensajes de éxito o error sin necesidad de salir de la página.
 - **Adaptación a dispositivos móviles.** Monitorea los eventos de expansión y colapso de la barra de navegación (`navbar`) de Bootstrap, ajustando la visibilidad del menú inferior para evitar que los controles oculten el contenido en pantallas pequeñas, asegurando una experiencia coherente y usable.
 - **Mantenimiento del estado y sincronización.** Coordina el estado del usuario, sincroniza los datos con el backend, responde a notificaciones en tiempo real y proporciona una navegación fluida, todo ello sin depender de frameworks SPA pesados. Incluye utilidades propias como funciones para formatear cantidades, IBANs y tamaños de texto, que refuerzan la usabilidad y la consistencia visual de la aplicación.
- **RTP.js** se encarga de transformar la sección *Request-to-Pay* (RTP) de la aplicación de página única (*single-page application*, SPA) en un panel interactivo que opera en tiempo real. Este módulo integra la interfaz visual con el backend, gestiona eventos dinámicos y ofrece flujos de trabajo adaptados a los roles de los usuarios, todo ello mientras mantiene un diseño ligero y sin dependencias de frameworks externos. A continuación, se detallan sus principales funcionalidades: El archivo `RTP.js` se encarga de transformar la sección *Request-to-Pay* (RTP) de la aplicación de página única (*single-page application*, SPA) en un panel interactivo que opera en tiempo real. Este módulo integra la interfaz visual con el backend, gestiona eventos dinámicos y ofrece flujos de trabajo adaptados a los roles de los usuarios, todo ello mientras mantiene un diseño ligero y sin dependencias de frameworks externos. A continuación, se detallan sus principales funcionalidades:

- **Carga inicial de la interfaz.** Una vez que el DOM está completamente cargado, el módulo descarga el fragmento `RTP.html` y lo inserta en el contenedor `seccionRTP`. Posteriormente, invoca un inicializador que registra los *event listeners* necesarios para la interacción. Este enfoque mantiene el HTML de la interfaz separado del código JavaScript, lo que reduce el peso de la página principal y facilita el mantenimiento.
- **Gestión de notificaciones y roles.** `RTP.js` almacena en memoria un arreglo con todas las notificaciones recibidas durante la sesión del usuario. Utilizando la variable global `currentActorRole`, determina qué panel de acciones debe mostrarse (Beneficiario, PSP Beneficiario, PSP Pagador o Pagador) y oculta los demás. Esta lógica de visibilidad se aplica tanto al cargar la sección como al cambiar de pestaña, asegurando que la interfaz se adapte dinámicamente al rol del usuario, incluso si la sesión se reutiliza para diferentes perfiles.
- **Flujos de trabajo por rol.** El módulo encapsula los procesos de negocio mediante manejadores de eventos que varían según el rol del usuario:

- * *Beneficiario:* Intercepta el formulario “Crear RTP”, construye un objeto JSON con el IBAN, el importe y el identificador del actor, y lo envía a `POST /rtp`. La respuesta del servidor se muestra en un contenedor de retroalimentación.
- * *PSP Beneficiario:* Proporciona formularios para validar al beneficiario (`POST /rtp/<id>/validate-benef`) y enrutar la solicitud (`POST /rtp/<id>/route`).
- * *PSP Pagador:* Ofrece un formulario para validar los fondos del pagador.
- * *Pagador:* Incluye un formulario para tomar la decisión final (aceptar o rechazar la solicitud).

Cada acción se realiza mediante la API Fetch sin recargar la página, y los resultados se notifican visualmente al usuario. Además, los identificadores RTP pueden accionarse desde botones en la tabla de notificaciones, permitiendo a PSPs y pagadores responder con un solo clic, sin necesidad de completar formularios manualmente.

- **Sincronización en tiempo real.** El módulo instala *listeners* WebSocket para los eventos `rtp_created`, `rtp_routed`, `rtp_validated_payer` y `rtp_decision`. Cuando el backend emite uno de estos eventos, el cliente verifica si el rol conectado es el destinatario. Si lo es, añade el objeto RTP al arreglo de notificaciones y actualiza la tabla de la interfaz. El renderizado genera filas con el importe, el estado y un botón contextual que activa la acción correspondiente según el rol y el estado actual, manteniendo la interfaz sincronizada sin necesidad de solicitudes adicionales al servidor.
 - **Auditoría de transacciones.** Un botón “Mostrar logs” realiza una solicitud `GET /logs`, recupera el historial de transacciones firmado con SHA-256 y lo presenta en un listado colapsable. Esto permite al usuario revisar el ciclo de vida de cada solicitud de manera instantánea, ofreciendo una herramienta de auditoría clara y accesible.
- **RTP.html** es una plantilla HTML autónoma que el cliente carga dinámicamente cuando el usuario accede a la sección Request-to-Pay; reúne, en un único fragmento, los formularios de acción de los cuatro roles (Beneficiary, PSP Beneficiary, PSP Payer y Payer) y una tabla de notificaciones donde se van listando los cambios de estado recibidos en tiempo real, de modo que toda la interfaz específica del flujo RTP se mantiene separada del documento principal y puede insertarse o actualizarse sin recargar la página.

- **styles.css** es consumido por todas las páginas del frontend (por ejemplo, index.html y los fragmentos dinámicos como RTP.html) y su única misión es dotar de coherencia visual a la aplicación: define la paleta de colores (degradados morados y dorados), las tipografías (Montserrat y Great Vibes), los espaciados y sombras de los componentes (cards, inputs, botones), y aplica transiciones suaves (hover, fade-in) para mejorar la percepción de interactividad sin añadir ninguna lógica de negocio.

3.4 Emulación del prototipo *Request To Pay*

Para emular el flujo Request To Pay, se han configurado cuatro actores con roles y permisos claramente diferenciados: el beneficiario, el pagador y los PSP correspondientes a cada uno de ellos. Cada actor dispone de un conjunto específico de endpoints REST para ejecutar sus operaciones (crear, validar, enrutar y decidir sobre una petición RTP) y escucha eventos WebSocket que garantizan la notificación en tiempo real de cada cambio de estado. De esta manera, el sistema reproduce fielmente el ciclo completo de una solicitud de cobro, desde su emisión por parte del beneficiario hasta la decisión final del pagador, manteniendo la trazabilidad y la coherencia de los estados en el backend y el frontend.

Tal como se describió anteriormente, todos los participantes deben estar previamente registrados en el Registro OSM, que actúa como fuente única de verdad y coordina la relación entre clientes y sus proveedores de servicio de pago. Gracias a este repositorio central, el banco del pagador ya conoce a su cliente y, de forma análoga, el banco del beneficiario está al tanto de su cliente beneficiario. Cuando este último genera una solicitud RTP introduciendo el IBAN del pagador, el OSM suministra automáticamente la información del PSP correspondiente, de modo que la petición se dirige al servicio adecuado. Por tanto, en esta emulación se asume la inscripción previa de los actores en el OSM y no se modela el proceso de registro dentro del flujo RTP.

Cuando ejecuto el código y me conecto al servidor lo que se ve es la página del login. Esto emula el proceso de login de la página web o aplicación de un banco real, en la cual si no eres cliente, no puedes acceder.

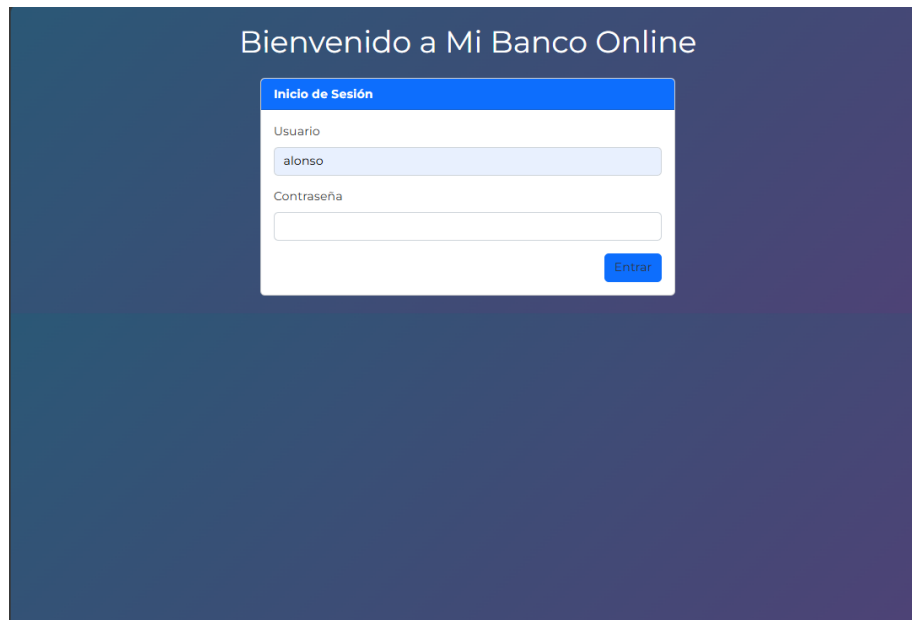


Figura 6: LoginForm

Tras iniciar sesión, la aplicación muestra la **página principal del banco** (Figura 7) con los datos esenciales del cliente —nombre, IBAN, saldo y fotografía— y un menú superior que da acceso a funcionalidades habituales como *Cuentas*, *Tarjetas* o *Perfil*. Esta interfaz reproduce la estética y el flujo de una banca en línea real: si no eres cliente, el proceso se detiene en la pantalla de inicio de sesión; si lo eres, se habilitan los paneles correspondientes a tus productos financieros.

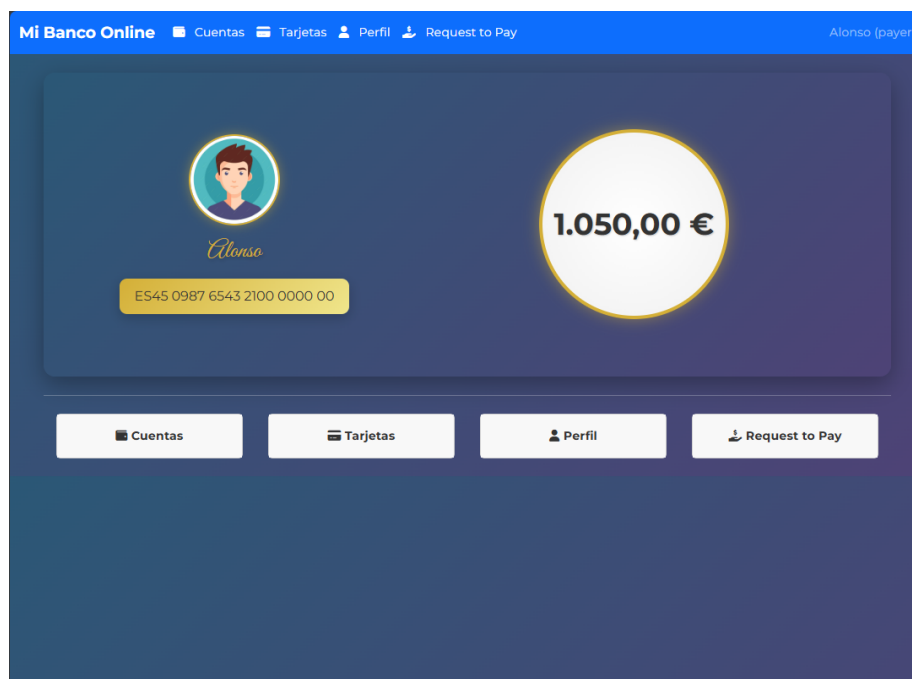


Figura 7: DashBoard

Lo verdaderammente novedoso del prototipo, y por tanto, el *producto* que se "vende" es la nueva opción *Request To Pay* que aparece en el menú. Dicha opción encapsula todo el software desarrollado ya explicado anteriormente.

La propuesta es que, una vez los bancos se adhieran al esquema RTP, todas las webs y aplicaciones bancarias incorporen esta sección tal y como se muestra aquí. El módulo es independiente, integrable vía API y compatible con las plataformas existentes, de manera que cualquier entidad que se registre en el sistema podría habilitarlo sin rediseñar su banca digital. En síntesis el producto de este TFG es un software que materializa el estándar RTP y lo hace accesible al usuario final desde la misma pantalla donde hoy consulta su saldo o sus tarjetas.

Como el objeto de interés es el apartado RTP el resto de apartados son secciones vacías ya que no aportan nada:

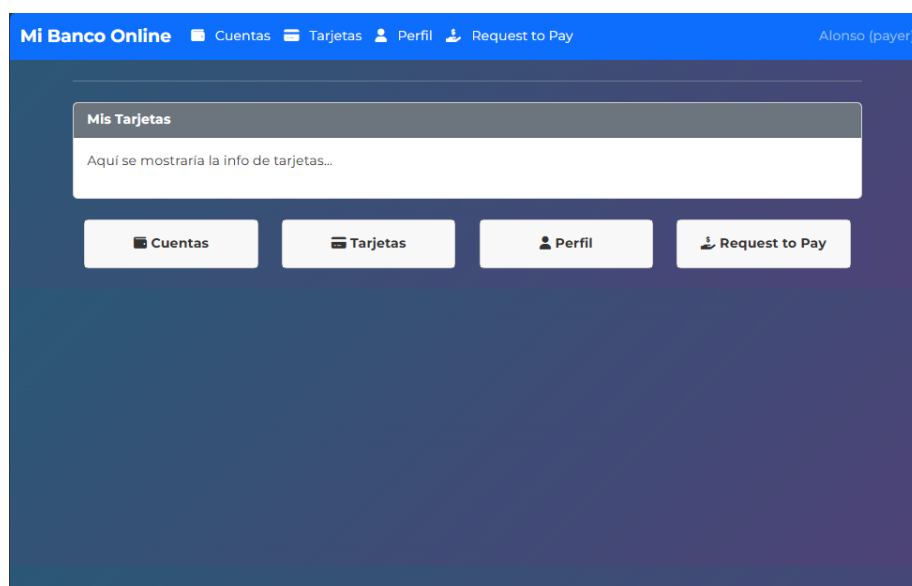


Figura 8: Tarjetas

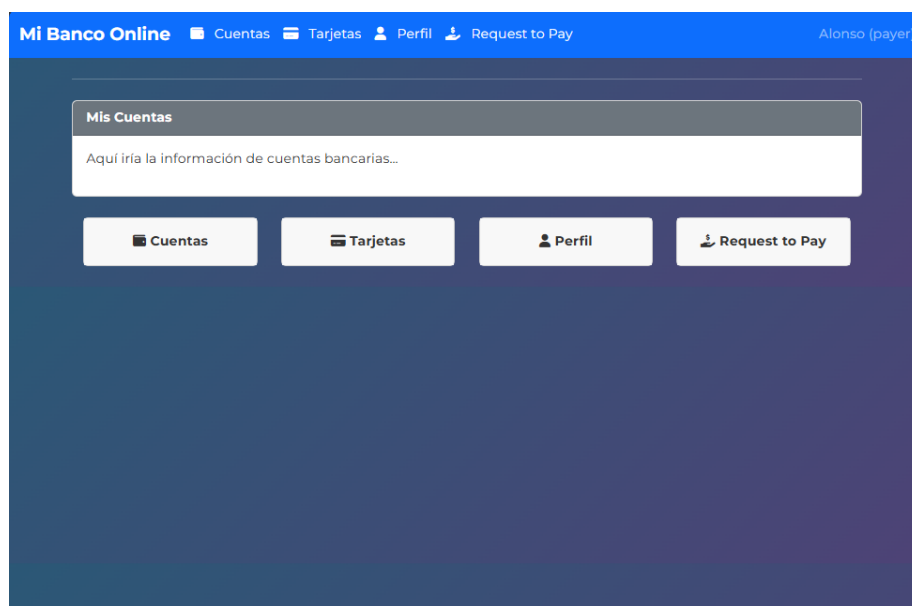


Figura 9: Cuentas

En la opción Perfil se pueden consultar o modificar algunos datos.

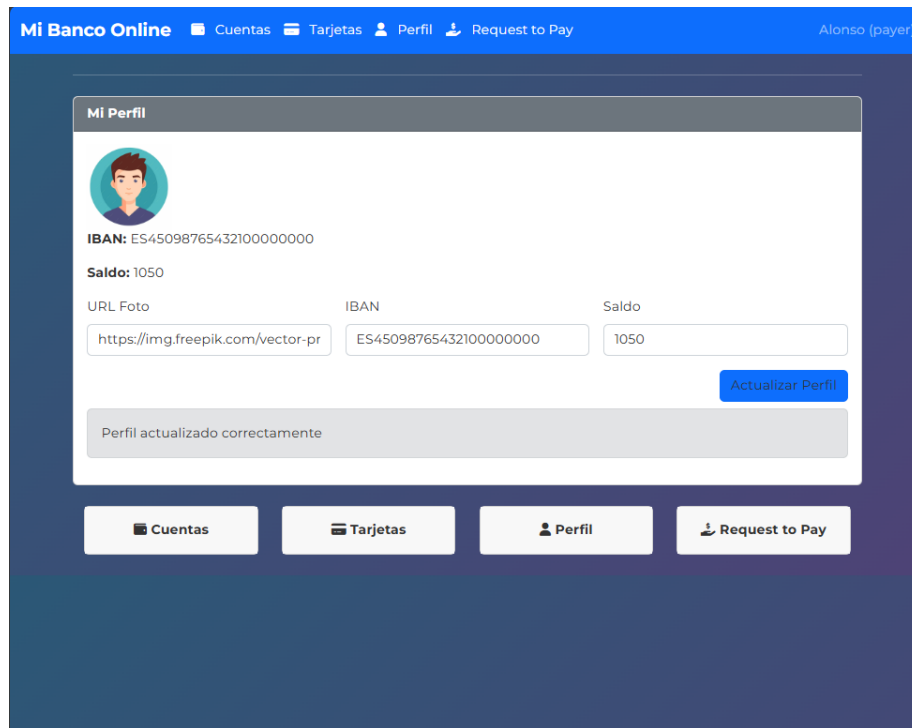


Figura 10: Perfil

Esto aplica a todos los usuarios registrados en la página del banco, tanto empresas como clientes.

A continuación voy a explicar cómo se modela el flujo de un proceso de pago mediante RTP. En primer lugar he iniciado sesión en la página del banco de los cuatro actores para ser conscientes de qué es lo que vería cada uno de ellos durante el proceso.

Me centraré en explicar principalmente el backend, ya que el frontend es una página web y no creo que haya que detallar demasiado.

- **Creación del RTP**

El flujo comienza cuando el beneficiario, ya autenticado, envía una petición POST a la ruta `/rtp`. A partir de aquí intervienen dos piezas de código: la capa de exposición HTTP y la capa de negocio que materializa la petición.

```

1      @rtp_blueprint.route('/rtp', methods=['POST'])
2      @role_required('beneficiary')
3      def crear_rtp():
4          data = request.get_json()
5          # ID del beneficiario que est  logueado
6          beneficiary_id = data.get('actor_id')
7          # IBAN del pagador
8          payer_iban = data.get('payer_iban')
9          # Importe solicitado
10         amount = data.get('amount')
```

```

11
12         if not payer_iban:
13             return jsonify({"error": "Falta iban"}), 400
14
15         if not amount:
16             return jsonify({"error": "Falta amount"}), 400
17
18         # Delegaci n a la capa de negocio
19         result = crear_rtp_service(data)
20         status = 201 if "error" not in result else 400
21         return jsonify(result), status

```

Listing 11: Endpoint creaci3n de solicitud RTP

1. Se impone un control de acceso que restringe la ruta a beneficiarios.
2. Se extraen y validan los tres datos imprescindibles: identificador del beneficiario, IBAN del pagador e importe.
3. Si alguno falta, se aborta el flujo devolviendo un error de validaci3n.
4. Con los datos verificados, la l3gica de negocio toma el relevo y, al finalizar, la ruta convierte su resultado en una respuesta HTTP adecuada (3xito o error).

```

1
2     def crear_rtp_service(data):
3         beneficiary_id = data.get('actor_id')
4         payer_iban     = data.get('payer_iban')
5         amount         = data.get('amount')
6
7         # 1) Validar beneficiario y su PSP
8         beneficiary = Actor.query.get(beneficiary_id)
9         if not beneficiary or beneficiary.role != 'beneficiary':
10             return {"error": "El actor no es beneficiario o no existe"}
11
12         if not beneficiary.psp_id:
13             return {"error": "El beneficiario no tiene PSP asociado"}
14         psp_benef_id = beneficiary.psp_id
15
16         # 2) Localizar pagador y su PSP
17         payer = Actor.query.filter_by(iban=payer_iban,
18                                     role='payer').first()
19         if not payer:
20             return {"error": "No se encontr un pagador con ese IBAN"}
21
22         if not payer.psp_id:
23             return {"error": "El pagador no tiene PSP asociado"}
24         psp_payer_id = payer.psp_id
25
26         # 3) Crear y guardar el RTP
27         nuevo_rtp = RTP(
28             iban=payer_iban,
29             amount=amount,
30             beneficiary_id=beneficiary.id,
31             psp_beneficiary_id=psp_benef_id,
32             psp_payer_id=psp_payer_id,

```

```

32         payer_id=payer.id
33     )
34     db.session.add(nuevo_rtp)
35     db.session.commit()
36
37     # 4) Notificación en tiempo real al PSP del beneficiario
38     socketio.emit(
39         'rtp_created',
40         nuevo_rtp.to_dict(),
41         room=f'psp_beneficiary_{psp_benef_id}'
42     )
43
44     return {
45         "message": "RTP creado correctamente",
46         "id": nuevo_rtp.id
47     }

```

Listing 12: Servicio de negocio creacion RTP

1. Se verifica que el beneficiario existe, posee el rol correcto y tiene un PSP asignado.
2. Se identifica al pagador mediante su IBAN y se confirma que también cuenta con un PSP registrado.
3. Una vez validados ambos actores y sus PSP, se crea un registro RTP que deja constancia de todos los datos relevantes y se persiste en la base de datos.
4. Al confirmarse la grabación, se emite un evento en tiempo real dirigido al canal del PSP del beneficiario, de modo que este recibe instantáneamente la solicitud y puede empezar a procesarla.

Gracias a la coordinación entre estos dos bloques de código el primer paso del flujo Request To Pay queda cubierto: la petición se registra de forma duradera y el PSP del beneficiario obtiene la información necesaria para continuar con las etapas siguientes.

- **Validación y enrutamiento del RTP por parte del PSP beneficiario**

Una vez que el beneficiario ha creado el RTP y el PSP del beneficiario ha recibido la notificación, este PSP asume el control para verificar la solicitud y, si todo es correcto, remitirla al PSP del pagador.

```

1         @rtp_blueprint.route('/rtp/<int:rtp_id>/validate-beneficiary',
2             methods=['POST'])
3         @role_required('psp_beneficiary')
4         def validar_beneficiario(rtp_id):
5             result = validar_beneficiario_service(rtp_id)
6             status = 200 if "error" not in result else 400
7             return jsonify(result), status
8
9         @rtp_blueprint.route('/rtp/<int:rtp_id>/route', methods=['POST'])
10        @role_required('psp_beneficiary')
11        def enrutar_rtp(rtp_id):
12            result = enrutar_rtp_service(rtp_id)
13            status = 200 if "error" not in result else 400

```

```
13         return jsonify(result), status
```

Listing 13: Endpoints para validación y enrutado

1. *Control de acceso.* Ambas rutas están protegidas para garantizar que sólo un actor con rol de PSP del beneficiario pueda ejecutar estas acciones.
2. *Validación inicial.* La primera ruta confirma que el RTP cumple los requisitos mínimos (coherencia de datos, existencia, formato, seguridad). Si la verificación falla, se devuelve un error; en caso contrario, el proceso avanza.
3. *Enrutamiento.* La segunda ruta, accesible tras la validación, pone el RTP en camino hacia el PSP del pagador, devolviendo una respuesta de éxito cuando la operación se completa o un error si surge alguna incidencia.

```
1         def validar_beneficiario_service(rtp_id):
2             rtp_obj = RTP.query.get(rtp_id)
3             if not rtp_obj:
4                 return {"error": "RTP no encontrado"}
5             result = cambiar_estado_rtp(db, rtp_obj,
6                                     "validado-beneficiario")
7
8             socketio.emit(
9                 'rtp_validated_beneficiary',
10                 rtp_obj.to_dict(),
11                 room=f'psp_beneficiary_{rtp_obj.psp_beneficiary_id}'
12             )
13             return result
14
15         def enrutar_rtp_service(rtp_id):
16             rtp_obj = RTP.query.get(rtp_id)
17             if not rtp_obj:
18                 return {"error": "RTP no encontrado"}
19             result = cambiar_estado_rtp(db, rtp_obj, "enrutado")
20
21             socketio.emit(
22                 'rtp_routed',
23                 rtp_obj.to_dict(),
24                 room=f'psp_payer_{rtp_obj.psp_payer_id}'
25             )
26             return result
```

Listing 14: Servicio de negocio validación y enrutado

1. *Recuperación y comprobación.* Para cada operación se localiza el registro RTP mediante su identificador y se verifica su existencia; de no hallarse, se informa del error.
2. *Actualización de estado.*
 - En la validación se marca el RTP como **validado-beneficiario**.
 - En el enrutamiento se actualiza a **enrutado**, indicando que ya ha sido despachado hacia el PSP del pagador.

3. *Persistencia.* Cada cambio de estado se registra de forma duradera en la base de datos, lo que asegura la trazabilidad del flujo.
4. *Notificación en tiempo real.* Tras la validación se emite un evento dirigido a la sala del propio PSP beneficiario para confirmar la operación; después, al enrutarse, se envía otro evento a la sala del PSP del pagador, de modo que este último reciba de inmediato la solicitud y pueda continuar con el proceso.

Con esta secuencia de pasos, el PSP del beneficiario garantiza la integridad técnica y comercial del RTP y lo traslada, de forma segura y casi instantánea, al siguiente eslabón de la cadena: el PSP del pagador.

- **Validación del RTP por el PSP del pagador**

Tras recibir el mensaje «enrutado» proveniente del PSP del beneficiario, el PSP del pagador debe decidir si la solicitud es aceptable desde un punto de vista operativo (fondos, reglas de riesgo, etc.). Esta responsabilidad vuelve a dividirse entre la capa HTTP y la capa de negocio.

```
1 @rtp_blueprint.route('/rtp/<int:rtp_id>/validate-payer',
2     methods=['POST'])
3 @role_required('psp_payer')
4 def validar_payer(rtp_id):
5     result = validar_payer_service(rtp_id)
6     status = 200 if "error" not in result else 400
7     return jsonify(result), status
```

Listing 15: Endpoint validación pagador

1. El acceso queda restringido al rol *psp_payer*, impidiendo que cualquier otro actor invoque la ruta.
2. Se delega la lógica de validación al servicio interno, traduciendo después su resultado a una respuesta HTTP de éxito (200) o error (400).

```
1 def validar_payer_service(rtp_id):
2     """
3     Validación por el PSP del pagador.
4     1) Si el pagador no dispone de fondos suficientes, se fuerza
5     el rechazo.
6     2) En caso contrario, se marca el RTP como validado por el PSP
7     del pagador.
8     """
9     rtp_obj = RTP.query.get(rtp_id)
10    if not rtp_obj:
11        return {"error": "RTP no encontrado"}
12
13    # Buscar al pagador asociado y verificar su saldo
14    from models import Actor
15    payer_actor = Actor.query.get(rtp_obj.payer_id)
16    if not payer_actor:
```

```

15         return {"error": "El payer asignado a este RTP no existe"}
16
17     if payer_actor.balance < rtp_obj.amount:
18         # Rechazo inmediato por falta de fondos
19         return rechazar_rtp(db, rtp_obj, "Saldo insuficiente (PSP
           forz cancelaci n)")
20
21     # Fondos suficientes implica validaci n satisfactoria
22     result = cambiar_estado_rtp(db, rtp_obj, "validado_payer")
23
24     # Notificaci n al pagador (front-end) para que tome su
           decisi n final
25     socketio.emit('rtp_validated_payer',
26                   rtp_obj.to_dict(),
27                   room=f'payer_{rtp_obj.payer_id}')
28
29     return result

```

Listing 16: Servicio de negocio validaci3n pagador

1. Se localiza el RTP y se verifica que existe en la base de datos.
2. Se consulta al actor *payer* para comprobar que realmente dispone de los fondos necesarios.
3. Si el saldo es insuficiente, el flujo se interrumpe y el RTP se marca como *rechazado*, devolviendo la causa concreta.
4. Cuando los fondos son suficientes, el estado cambia a *validado_payer*. Simultáneamente se emite un evento en tiempo real hacia la sala del pagador, de modo que su aplicaci3n cliente muestre la notificaci3n y solicite al usuario la decisi3n definitiva (aceptar o rechazar el cobro).

Con estas dos piezas de c3digo el PSP del pagador completa su parte: revisa la viabilidad financiera y, si procede, deja el RTP listo para que el pagador humano confirme el pago o lo rechace.

• Decisi3n final del pagador

Cuando el PSP del pagador ha declarado la solicitud viable, el control pasa al propio pagador, que puede aceptar o rechazar el cobro. Este 3ltimo paso del flujo se implementa, como los anteriores, con una ruta HTTP y una pieza de l3gica de negocio.

```

1     @rtp_blueprint.route('/rtp/<int:rtp_id>/decision',
           methods=['POST'])
2     @role_required('payer')
3     def decision_payer(rtp_id):
4         data = request.get_json()
5         result = decision_payer_service(rtp_id, data)
6         status = 200 if "error" not in result else 400
7         return jsonify(result), status

```

Listing 17: Endpoint decisi3n final

1. El acceso se restringe al rol *payer*; ningún otro actor puede invocar la ruta.
2. La decisión (aceptar o rechazar) llega en el cuerpo JSON junto con la identidad del pagador.
3. El resultado se traduce en una respuesta HTTP con código 200 si todo fue correcto o 400 si se produjo un error de validación.

```

1  def decision_payer_service(rtp_id, data):
2      """
3      Decisi n final del pagador: 'aceptado' o 'rechazado'.
4      - Si es 'aceptado', se descuentan los fondos.
5      - Se registra el nuevo estado y se notifica al beneficiario.
6      """
7      rtp_obj = RTP.query.get(rtp_id)
8      if not rtp_obj:
9          return {"error": "RTP no encontrado"}
10
11     decision = data.get('decision')
12     if decision not in ["aceptado", "rechazado"]:
13         return {"error": "Decisi n no v lida"}
14
15     # Descontar fondos si procede
16     if decision == "aceptado":
17         from models import Actor
18         payer_actor = Actor.query.get(rtp_obj.payer_id)
19         if not payer_actor:
20             return {"error": "El payer asignado no existe"}
21         if payer_actor.balance < rtp_obj.amount:
22             return rechazar_rtp(db, rtp_obj,
23                                 "Saldo insuficiente en el ltimo
24                                 momento")
25         payer_actor.balance -= rtp_obj.amount
26         db.session.commit()
27
28     result = cambiar_estado_rtp(db, rtp_obj, decision)
29
30     # Aviso en tiempo real al beneficiario
31     socketio.emit('rtp_decision',
32                  rtp_obj.to_dict(),
33                  room=f'beneficiary_{rtp_obj.beneficiary_id}')
34
35     return result

```

Listing 18: Servicio de negocio decisi3n final

1. Se recupera el RTP y se verifica que existe; en caso contrario, el proceso se detiene con un error.
2. Se comprueba que la elecci3n recibida sea «aceptado» o «rechazado». Cualquier otro valor provoca un error de validaci3n.
3. Si la opci3n es «aceptado», se confirma que el pagador dispone de saldo suficiente y, de ser as3, se descuenta el importe correspondiente de su cuenta. Si no hay fondos, el RTP se marca inmediatamente como rechazado.

4. Finalmente se actualiza el estado del RTP con la decisión tomada y se registra un log con traza hash; al mismo tiempo se envía una notificación en tiempo real al beneficiario para informarle del resultado.

Con estas acciones concluye el flujo Request To Pay: el pagador emite su veredicto, se ejecuta (o cancela) el cargo y todas las partes quedan notificadas del desenlace.

3.5 Pruebas y validación

Para las pruebas en *Postman* lo primero fue definir la variable de entorno **baseURL**, que almacena la dirección raíz del servidor (Fig. 11). De este modo evito repetir la URL completa en cada petición POST o GET.

	Variable	Initial value	Current value
<input checked="" type="checkbox"/>	base_url	http://127.0.0.1:5000	http://127.0.0.1:5000
	Add new variable		

Figura 11: Variable **baseURL** en Postman

Con la variable configurada, diseñé un flujo mínimo de peticiones para verificar los principales endpoints de la API. Al inicio del proyecto el servidor carecía de interfaz web, por lo que toda la validación se realizó a través de Postman: definía la petición, añadía los parámetros necesarios y comprobaba la respuesta. Este ciclo de prueba-error permitió depurar la lógica hasta conseguir un comportamiento estable.

Durante las primeras fases existían más endpoints de los que finalmente se han publicado; algunos, como la creación de actores, siguen disponibles pero no están expuestos en la aplicación web.

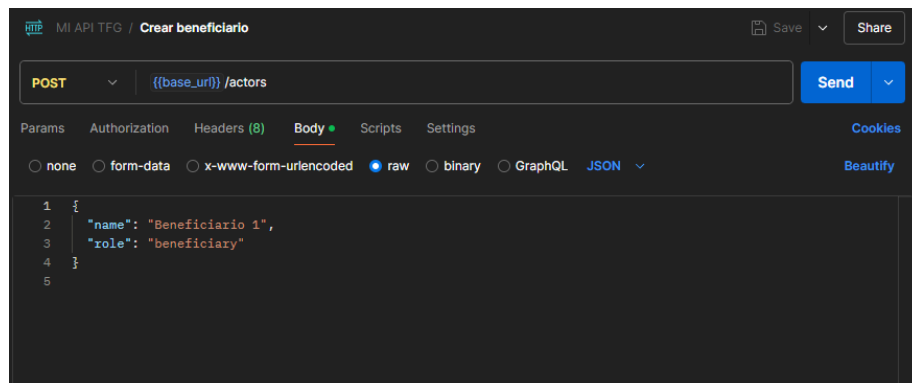


Figura 12: crearActor

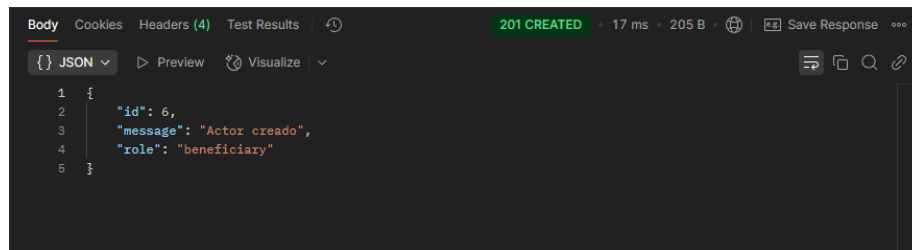


Figura 13: crearActorResponse

Durante las primeras fases del desarrollo se creó una colección de peticiones que se activa manualmente para verificar el correcto funcionamiento de la API tras cada cambio en las funcionalidades:

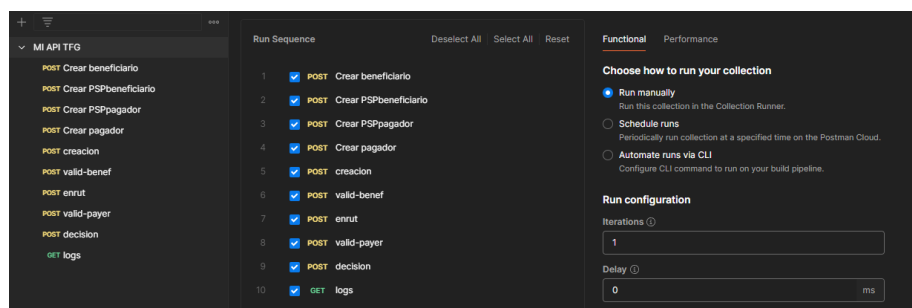


Figura 14: MiAPITFG

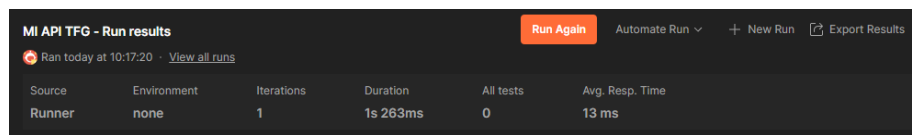


Figura 15: RunResults

Las peticiones relevantes del flujo son:

1. Creación RTP

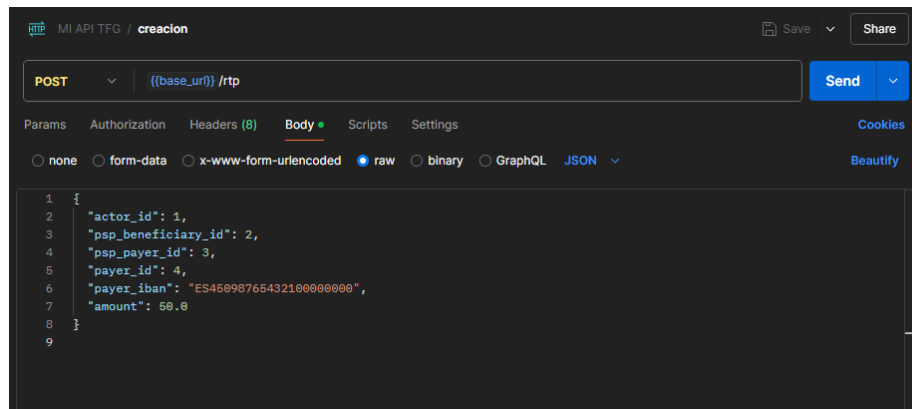


Figura 16: crearRTP

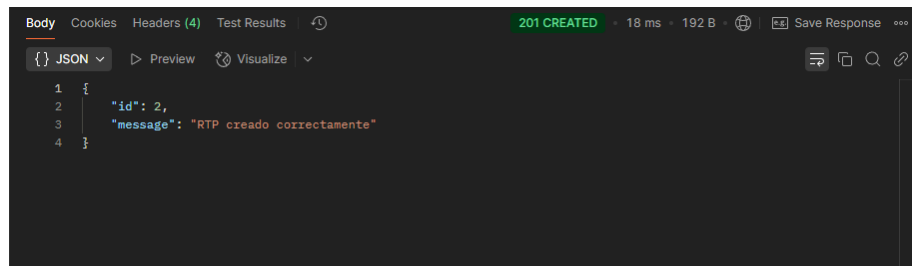


Figura 17: crearRTPResponse

2. Validación y enrutado del PSP del beneficiario

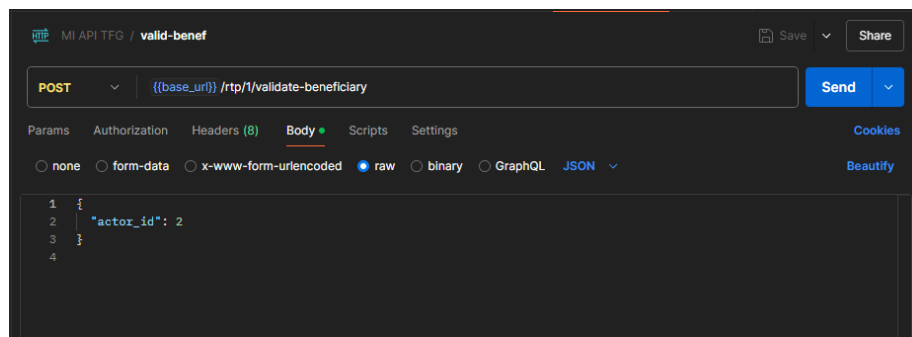


Figura 18: validarBenef

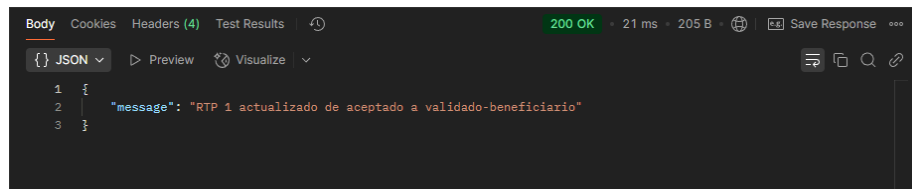


Figura 19: validarBenefResponse

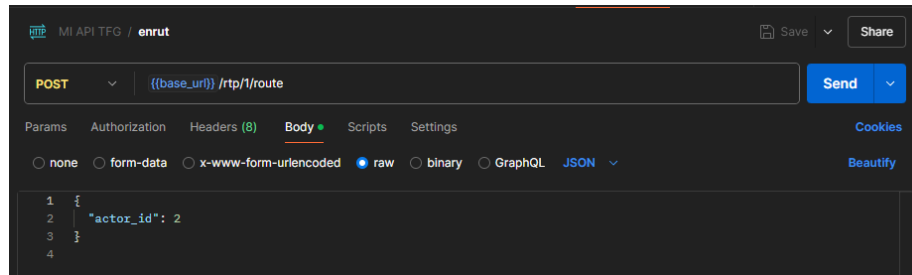


Figura 20: enrutarBenef

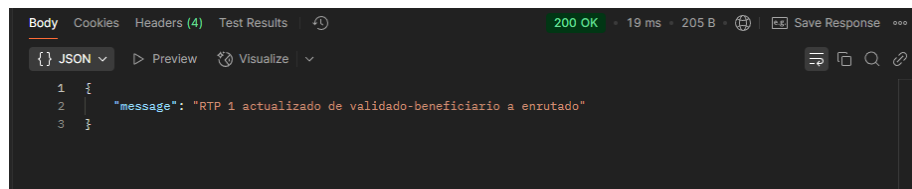


Figura 21: enrutarBenefResponse

3. Validación del PSP del pagador

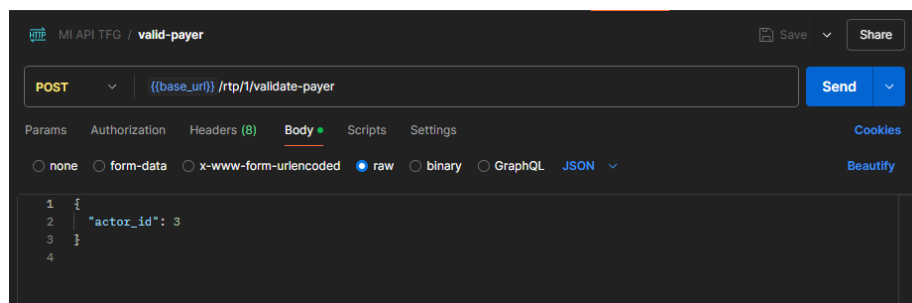


Figura 22: validarPayer

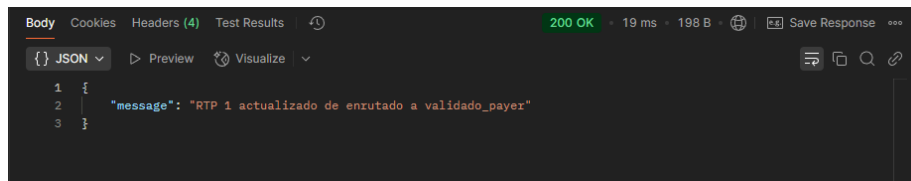


Figura 23: validarPayerResponse

4. Decisión final

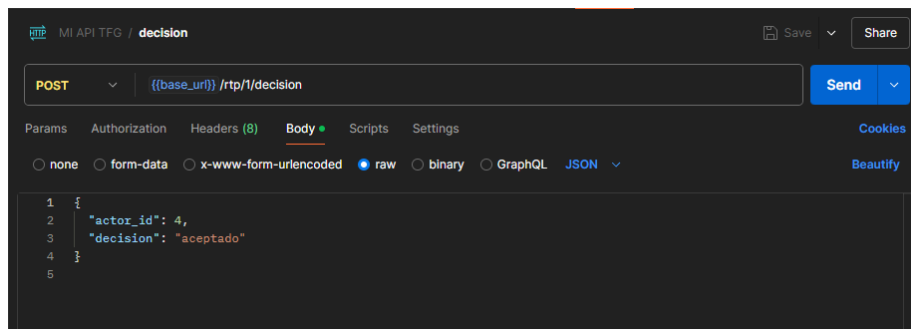


Figura 24: decision

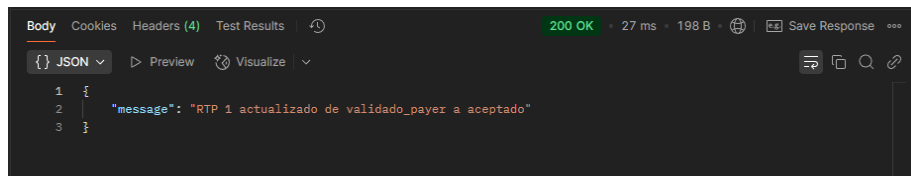


Figura 25: decisionResponse

5.

Hay ocasiones en las que el flujo no va a completarse debido a errores o rechazos:

Si al crear el RTP el IBAN no corresponde con ningún cliente registrado en el esquema RTP o no cumple la normativa ISO 13616¹⁹

¹⁹Esta norma internacional define la estructura del IBAN: ES + 2 dígitos de control + 20 dígitos del número de cuenta

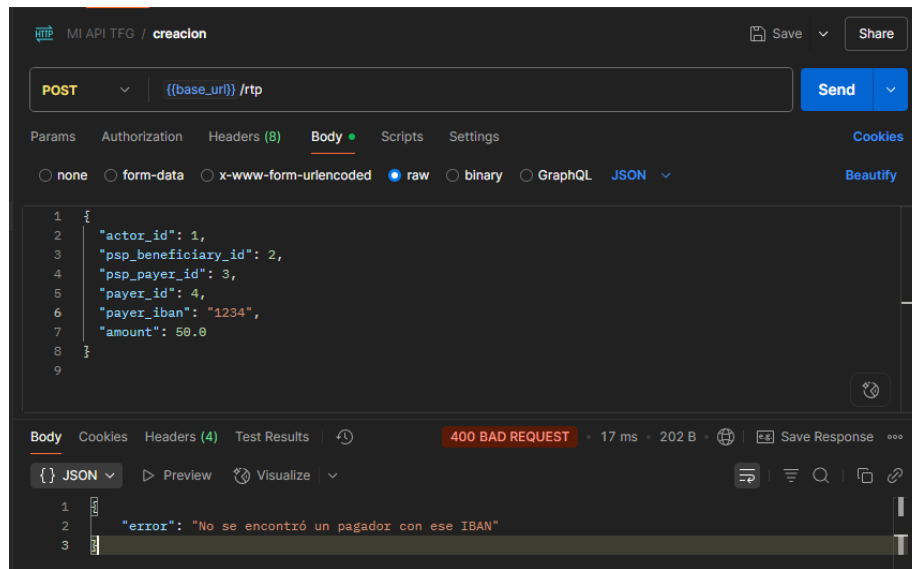


Figura 26: error1

Si se crea un RTP con importe mayor al que el pagador puede hacer frente, el RTP llegará hasta el PSP del pagador, quien tras comprobar el importe, lo rechazará.

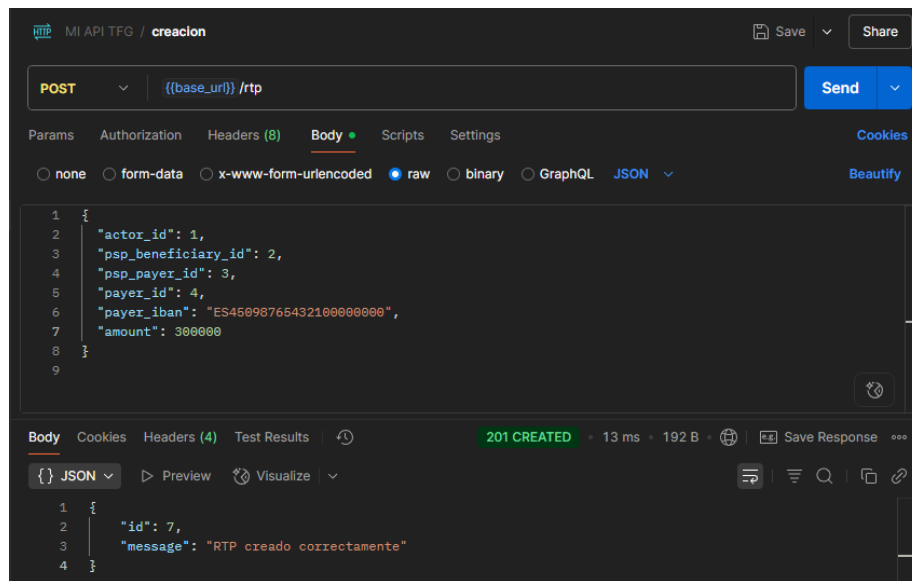


Figura 27: error2_1

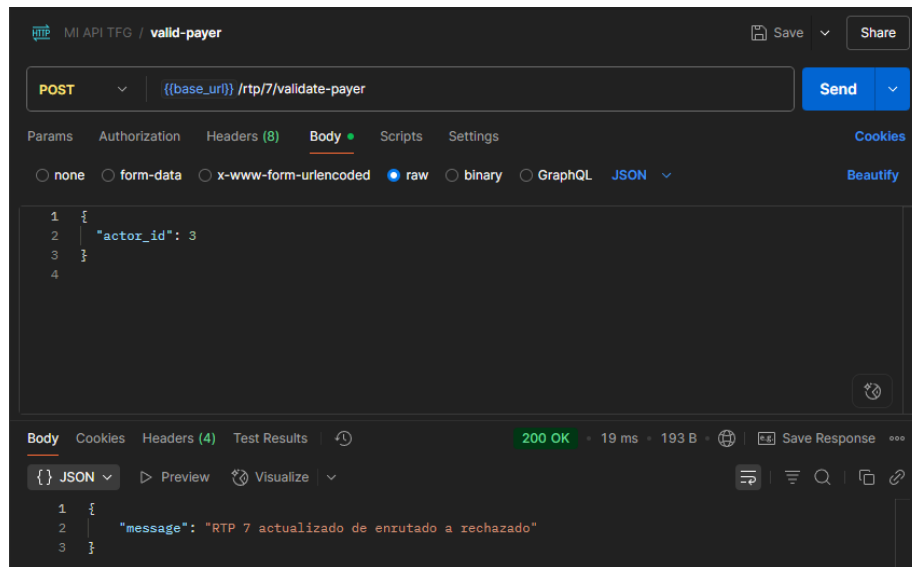


Figura 28: error2_2

Cuando se crea un RTP deben existir los cuatro actores involucrados o de lo contrario también dará un error:

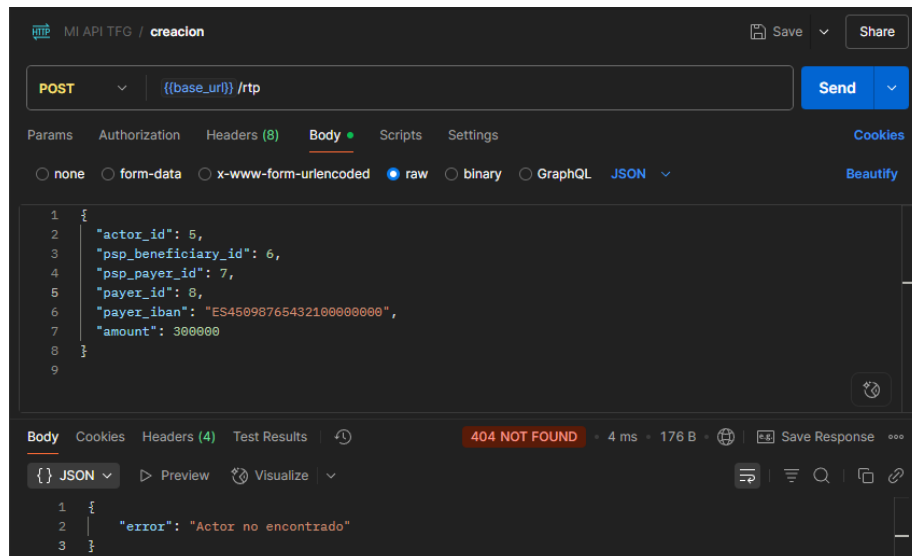


Figura 29: error3

4 Conclusiones y líneas futuras

El desarrollo de este Trabajo de Fin de Grado (TFG) ha representado un esfuerzo significativo que ha culminado en la creación de un prototipo funcional de un servidor HTTP basado en el esquema Request To Pay (RTP), una iniciativa impulsada por el European Payments Council (EPC) para modernizar y agilizar los procesos de cobro en Europa. El objetivo principal del proyecto —diseñar e implementar un software que simulara las operaciones fundamentales del esquema SEPA RTP utilizando tecnologías actuales y accesibles— se ha alcanzado con éxito, lo que supone un logro tanto técnico como académico. Este trabajo no solo me ha permitido poner en práctica los conocimientos adquiridos durante mi formación en telecomunicaciones y desarrollo de software, sino también explorar un ámbito tan relevante como los sistemas de pago digitales, que desempeñan un papel crucial en la economía global de hoy.

A nivel personal, este TFG ha sido una buena oportunidad para profundizar en el ecosistema SEPA y sus diferentes esquemas de pago. En particular, he podido analizar las limitaciones del SDD, como su dependencia de plazos largos y su falta de flexibilidad para ciertos casos de uso, y contrastarlas con las ventajas que ofrece RTP, como la inmediatez en las transacciones y la mejora en la interacción entre pagadores y beneficiarios. Este aprendizaje no se ha limitado al ámbito teórico: la implementación práctica del prototipo me ha enfrentado a retos técnicos que han fortalecido mis competencias como desarrollador. Por ejemplo, integrar WebSockets para permitir una comunicación bidireccional en tiempo real entre los actores del sistema, diseñar una base de datos eficiente para almacenar y gestionar el estado de las solicitudes, y crear una interfaz de usuario sencilla pero funcional han sido tareas que me han exigido creatividad, paciencia y un enfoque metódico.

El proceso de desarrollo también ha sido una lección sobre la importancia de una metodología bien definida, lo que me permitió avanzar de manera constante, detectar errores a tiempo y ajustar los objetivos según las necesidades del proyecto. Herramientas como Postman, que utilicé para probar exhaustivamente la API del servidor, y GitHub Actions, que automatizó la integración continua, no solo facilitaron el desarrollo, sino que también me introdujeron a prácticas profesionales que son esenciales en la industria del software. Este enfoque estructurado, combinado con una documentación detallada de cada etapa, ha sido clave para mantener el control sobre el proyecto.

El sistema RTP se posiciona como un candidato ideal para convertirse en un estándar en la zona SEPA, especialmente a medida que más instituciones financieras y PSP adopten el esquema y lo integren en sus operaciones. En el futuro, es probable que RTP no solo facilite los pagos entre empresas y consumidores, sino que también fomente una mayor interoperabilidad y eficiencia en el mercado financiero europeo, contribuyendo a una economía más conectada y dinámica.

4.1 Potencial y aplicación real

En cuanto al prototipo desarrollado, aunque satisface plenamente los objetivos establecidos para este TFG, su potencial va mucho más allá de un simple ejercicio académico. Durante el proceso identifiqué varias áreas de mejora que podrían transformar este simulador en una herramienta aplicable en escenarios reales. A continuación, detallo algunas de estas oportunidades de evolución:

1. **Escalabilidad del sistema:** La base de datos SQLite, aunque suficiente para un prototipo, tiene limitaciones en términos de rendimiento y capacidad. Para soportar un mayor volumen de usuarios y transacciones, sería necesario migrar a un sistema más robusto como PostgreSQL o MySQL, que ofrecen mejor escalabilidad y soporte para entornos de producción.

2. **Fortalecimiento de la seguridad:** El prototipo incluye medidas básicas de autenticación, pero un sistema real requeriría estándares más altos, como la encriptación de extremo a extremo, autenticación multifactor y cumplimiento con normativas como PSD2 (Payment Services Directive 2) y GDPR (General Data Protection Regulation). Estas mejoras garantizarían la protección de los datos sensibles y la confianza de los usuarios.
3. **Interoperabilidad con sistemas bancarios:** Para que el prototipo trascienda su estado actual, sería esencial integrarlo con las APIs de bancos y PSP reales. Esto permitiría ejecutar transacciones monetarias auténticas y demostrar su utilidad en un contexto práctico, un paso crítico hacia su adopción en el mundo real.
4. **Ampliación de funcionalidades:** El sistema podría enriquecerse con características avanzadas, como soporte para pagos recurrentes (ideal para suscripciones o facturas periódicas), compatibilidad con múltiples monedas (facilitando transacciones transfronterizas), herramientas analíticas que ofrezcan estadísticas a los usuarios, y reportes detallados sobre el historial de pagos. Estas adiciones harían que el sistema fuera más versátil y atractivo para distintos tipos de usuarios.
5. **Mejora de la experiencia de usuario:** Aunque el frontend actual es funcional, podría optimizarse con un diseño más moderno y accesible. Incorporar elementos como notificaciones push, un historial visual de transacciones, opciones de personalización y una interfaz adaptada a dispositivos móviles elevaría la usabilidad y la satisfacción del usuario.

Estas mejoras, aunque son algo ambiciosas, son alcanzables con el tiempo y los recursos adecuados. Implementarlas no solo incrementaría la funcionalidad del prototipo, sino que también lo alinearía con las demandas de un mercado financiero en constante cambio, donde la innovación y la adaptabilidad son esenciales.

Más allá de los aspectos técnicos, este TFG ha sido una experiencia profundamente formativa a nivel personal y profesional. Me ha enseñado a gestionar proyectos complejos, a resolver problemas de manera creativa y a valorar la importancia de la perseverancia frente a los desafíos. También me ha abierto los ojos al enorme potencial de la tecnología para transformar sectores tradicionales como el financiero, un campo que, aunque puede parecer distante de las telecomunicaciones a primera vista, está intrínsecamente conectado a través de la infraestructura digital que lo sustenta. Esta intersección entre disciplinas es, en mi opinión, uno de los aspectos más fascinantes de este proyecto y una motivación para seguir explorando esta área en el futuro.

En conclusión, este Trabajo de Fin de Grado ha sido más que un requisito académico: ha supuesto un descubrimiento, aprendizaje y crecimiento. El esquema Request To Pay, con su enfoque innovador y su capacidad para simplificar los procesos de cobro, tiene el potencial de redefinir cómo interactuamos con los servicios financieros en Europa y más allá. El prototipo que he desarrollado, aunque es limitado en su alcance actual, es una prueba tangible de ese potencial. Con las mejoras adecuadas, podría evolucionar para convertirse en una solución práctica y relevante, contribuyendo a un ecosistema de pagos más eficiente, seguro y accesible. Estoy contento con el trabajo realizado y estoy seguro de que este sistema marcará el futuro en los pagos digitales.

Referencias

- [Cou23a] E. P. Council. EPC137 - SRTP Implementation Guidelines, 2023.
- [Cou23b] E. P. Council. EPC164 - API Security Framework, 2023.
- [Cou25] E. P. Council. SRTP Scheme Rulebook v4.0, 2025.

Índice de Figuras

1	4CornerModel	12
2	Layer	22
3	LayerComp	24
4	Tree	28
5	Arranque	29
6	LoginForm	38
7	DashBoard	39
8	Tarjetas	40
9	Cuentas	40
10	Perfil	41
11	Variable baseUrl en Postman	48
12	crearActor	48
13	crearActorResponse	49
14	MiAPITFG	49
15	RunResults	49
16	crearRTP	50
17	crearRTPResponse	50
18	validarBenef	50
19	validarBenefResponse	51
20	enrutarBenef	51
21	enrutarBenefResponse	51
22	validarPayer	51
23	validarPayerResponse	52
24	decision	52
25	decisionResponse	52
26	error1	53
27	error2_1	53
28	error2_2	54
29	error3	54

Índice de Tablas

1	Comparativa entre SDD y SRTP con SCT Inst	8
2	Pasos del flujo del esquema SEPA Request-to-Pay (SRTP) y su descripción	12
3	Pasos del esquema SRTP	15

Índice de Códigos

1	Configuración y arranque del servidor Flask-SocketIO	29
2	Configuración SQLAlchemy y SQLite	29
3	Modelo ORM + clase Actor	30
4	Funciones utilitarias	31
5	Decorador para control de funciones	31
6	Declaración blueprint rtp	32
7	Endpoint login	32
8	Endpoint profile	33
9	Declaración blueprint actores	33
10	Instancia global de socketIO	34
11	Endpoint creación de solicitud RTP	41
12	Servicio de negocio creacion RTP	42
13	Endpoints para validación y enrutado	43
14	Servicio de negocio validación y enrutado	44
15	Endpoint validación pagador	45
16	Servicio de negocio validación pagador	45
17	Endpoint decisión final	46
18	Servicio de negocio decisión final	47