Universidad de Valladolid

Escuela Técnica Superior de Ingenieros de Telecomunicación

Trabajo de Fin de Grado

Grado en Ingeniería de Tecnologías de Telecomunicación

Implementación de un Servidor HTTP para Emular un Prototipo del Sistema Request To Pay del EPC

Autor: Alonso
Sandoval Martínez
Tutores:
D. Tutor1
D. Tutor2

Valladolid, MES 202X

Índice

Agradecimientos

Agradezco a mis tutores, familia y amigos por su apoyo durante la realización de este trabajo. Su orientación y motivación han sido fundamentales para completar este proyecto.

Resumen

Este trabajo presenta la implementación de Como se muestra en [**?**], el protocolo SEPA permite... un servidor HTTP para emular un prototipo del sistema Request To Pay desarrollado por el European Payments Council (EPC), abordando...

Palabras clave— Servidor HTTP, Request To Pay, EPC, Telecomunicaciones, Pagos Electrónicos

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 detectado una serie de ineficiencias resumidas en los siguientes **5** puntos:

1. **Modelo off-line y necesidad de mandato físico**

   - El pago se inicia sin interacción *online*; El deudor debe firmar y remitir **mandato SEPA**[2] y el acreedor debe custodiarlo durante el periodo de vida del contrato y hasta 14 meses tras la última transacción.
   - No existe un estándar eMandate[3] interoperable, cada banco utiliza su propio estándar.

   Consecuencias: Fricción en la venta digital, costes de back-office (archivado, auditorías y gestión) y riesgo legal si el mandato no aparece ante una devolución.

[1] *SEPA Direct Debit* es el instrumento paneuropeo de cargo en cuenta regulado por el *European Payments Council*.
[2] Autorización que un deudor otroga a un acreedor para permitir el cobro automático de pagos mediante SDD.
[3] Un eMandate es la versión digital de un mandato.

2. **Derecho a devolución prolongado**

   - *Cobro autorizado*: el deudor dispone de **8 semanas** para devolver sin causa (*"no-questions-asked"*).
   - *Cobro no autorizado*: hasta **13 meses** para reclamar si el banco emisor no puede demostrar mandato válido.

   Consecuencias: gran incertidumbre sobre la firmeza del ingreso, reservas de liquidez, provisiones contables y alto nivel de *friendly fraud*[4].

3. **Ciclos de cobro lentos**

   - En esquema CORE[5]: envío al banco **D-5** para primera domiciliación y **D-2** para recurrencias; liquidación interbancaria **+2 días**.
   - 6–8 días naturales entre petición y abono firme, incompatibles con venta inmediata o entrega digital.

   Consecuencias: Tesorería imprevisible (cash-flow) y riesgo de prestar servicio sin cobro confirmado.

4. **Costes y complejidad de las R-transactions**[6]

   - Existen distintos códigos de R-transactions y cada uno de ellos implica un flujo distinto, lo que dificulta la gestión.

   Consecuencias: Necesidad de equipos específicos de conciliación y recobro, lo que conlleva un coste directo y pérdida de productividad.

5. **Ausencia de autorización fuerte (SCA)**[7]

   - El SDD se basa en consentimiento previo, no aplica SCA al momento del cargo.

   Consecuencias: Mayor riesgo de cargos disputados y se desaprovechan métodos de identidad digital ya existentes.

En conclusión. Estas ineficiencias se traducen en (i) estructura de costes elevada por devoluciones y personal especializado; (ii) liquidez incierta—los ingresos se confirman con días de retraso y pueden desaparecer meses después—y (iii) freno a la economía digital online, incapaz de ofrecer experiencias de pago instantáneas comparables a tarjeta, monederos o Bizum.

[4] servicio consumido y posterior devolución.

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

[6] transacciones de rechazo, devolución o reembolso asociadas a pagos fallidos o no autorizados.

[7] La SCA (Strong Customer Authentication) es un mecanismo de seguridad que exige verificar al usuario con al menos dos factores.

1.1.2 Oportunidad de un esquema Request-to-Pay

El estándar *SEPA Request-to-Pay* (**SRTP**)[8] se perfila como la evolución natural de la domiciliación SEPA. Sus ventajas técnicas frente al SDD son:

a) **Autenticación reforzada y consentimiento digital inmediato**
El acreedor emite un request que el deudor aprueba en su banca o *wallet* mediante SCA. Este gesto sustituye al mandato físico y genera una prueba electrónica de consentimiento, firmada y almacenada dentro del PSP del pagador.

b) **Irrevocabilidad y mitigación de fraude *post-servicio***
Tras la aceptación, el pago se realiza mediante SCT Inst[9]. Al no existir derecho de devolución automática, se elimina el *friendly fraud* asociado a la devolución de recibos y se reducen provisiones por impago.

c) **Liquidez *real-time* y conciliación automática**
La disponibilidad de fondos en $\leq 10\,\text{s}$ permite planificar tesorería al instante. Los identificadores de extremo a extremo y las referencias estructuradas ISO 20022 se transmiten sin perderse entre sistemas, de modo que la conciliación queda totalmente automatizada.

d) **Simplificación operativa**
Desaparecen las R-TRANSACTIONS, la custodia de mandatos y las tareas de back-office. El flujo se limita a dos mensajes (request y aceptación) y, opcionalmente, una transferencia instantánea, con clara trazabilidad extremo a extremo.

e) **Flexibilidad comercial y costes reducidos**
SRTP admite cobros únicos, recurrentes y fraccionados desde web o app vía enlaces profundos, QR o API, y al ser un pago mediante SCT Inst, las comisiones bancarias son muy inferiores a las de tarjeta o a las de gestión de devoluciones SDD.

En síntesis, SRTP traslada las ventajas históricas de la domiciliación—bajo coste y cobertura paneuropea—al entorno digital e inmediato, resolviendo los puntos técnicos que hoy limitan la competitividad del SDD en España y en la zona SEPA.

[8] Iniciativa del European Payments Council que define un flujo de solicitud (*request*) y aceptación de pagos en tiempo real, apoyado en mensajería **ISO 20022** (p. ej. `pain.013/pain.014`) y agnóstico respecto al instrumento de liquidación posterior.

[9] SCT INST: transferencia inmediata SEPA con liquidación 10 s.

Aspecto	SDD	SRTP (+ SCT Inst)
Autorización	Mandato off-line	Consentimiento digital (SCA) en tiempo real
Plazo de devolución	8 semanas / 13 meses	No aplica (operación irrevocable)
Disponibilidad de fondos	5–8 días	Segundos (<10 s)
Coste operativo	Alto (mandatos, R-CODES)	Bajo (mensajería ISO 20022, sin excepciones)
Cobertura *e-commerce*	Limitada	Óptima (API / móvil)
Riesgo de fraude	Medio-Alto (devolución)	Bajo (SCA + irreversibilidad)

1.2 Objetivos

El **objetivo general** del Trabajo Fin de Grado es entregar un *stack* de software que simule un proveedor *end-to-end* del esquema *SEPA Request-to-Pay* (SRTP), alineado con la versión 4.0 del *SRTP Scheme Rulebook* [**?**] y las guías técnicas del EPC (*EPC137* y *EPC164*) [**?**, **?**]. El prototipo debe ser instalable con `Docker Compose`, exponer una API HTTP/JSON conforme a *OpenAPI 3.1* y cubrir las cuatro operaciones core *(create, reject, response, cancel)* del flujo SRTP.
Para acotar y medir el trabajo se definen las siguientes **metas específicas**:

1. Implementar en `Node 20 LTS/Express` los *endpoints* REST de alta disponibilidad y su contraparte *callback* para notificaciones asíncronas (`Socket.IO`).
2. Desarrollar un módulo de firma, sellado temporal y validación X.509 (QSeal/QWAC) para garantizar requisitos de *identificación, autenticación y autorización* definidos por el *API Security Framework* [**?**].
3. Persistir estado y auditoría en una base `SQLite` ligera (`SQLAlchemy`) con modelo de datos alineado a los *datasets* DS-02, DS-07 y DS-10 del Rulebook.
4. Entregar una colección `Postman` y un `runner` CI (GitHub Actions) que ejecute casos de prueba de integración, incluyendo validación de esquemas JSON contra *schemas* oficiales del EPC.
5. Documentar las divergencias norma → implementación y proponer una hoja de ruta para su homologación futura en el *EDS*.

1.3 Fases y Métodos

Se adopta un ciclo **ágil**, con *sprints* de dos semanas y reuniones *review/retro*. Cada iteración termina con un incremento funcional desplegado en `Docker Hub` y su etiqueta asociada en `Git`.

Fase 1 – Análisis
- Lectura detallada de los documentos EPC 014, 137 y 164 y extracción de requisitos funcionales, de seguridad y de interoperabilidad.
- Modelado de actores en un diagrama de cuatro esquinas (Payee / Payer / $\text{PSP}_{\text{Payee}}$ / $\text{PSP}_{\text{Payer}}$), identificando puntos de confianza y certificados requeridos.

- Priorización de *user-stories* y definición de *Definition of Done*.

Fase 2 – Diseño e implementación

- Arquitectura `Clean Architecture` sobre `Express`: `routes.js`, `services.js`, `models.py`.
- Middleware de firma y verificación con `crypto.subtle` y librerías OpenSSL; generación de certificados de prueba `make cert`.
- Persistencia en `SQLite` mediante `Sequelize`; migraciones automáticas.
- WebSocket `Socket.IO` encapsulado en `ext_socketio.py` para notificaciones *push* de estado.

Fase 3 – Pruebas y validación

- Suite `Jest` + `supertest` para pruebas unitarias y de integración.
- Colección `Postman` con *scripts* pre/post-request que firman, estampan fecha y validan contra esquemas.
- Inyección de fallos con `toxiproxy`: retardos, caídas de red y respuestas 4xx/5xx para cubrir los flujos síncrono y asíncrono.
- Informe de cobertura y reporte SonarQube en el pipeline CI.

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 [**?**], SRTP related API Specifications v3.1 [**?**], API Security Framework v2.0 [**?**], 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

[10] ISO 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.

[11] ERPB: 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 20022[12], lo que facilita su integración con las plataformas SEPA existentes.

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

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

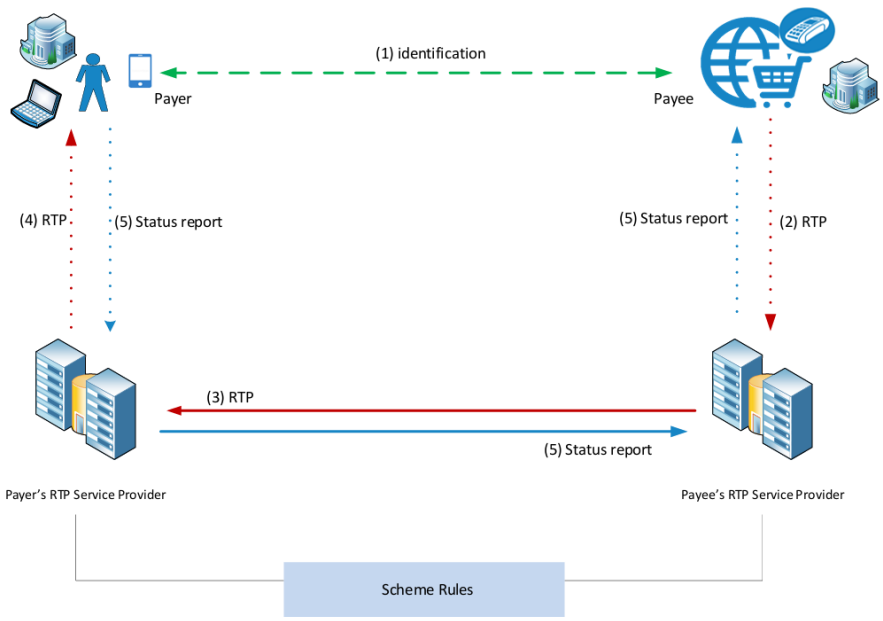


Figura 1: 4CornerModel

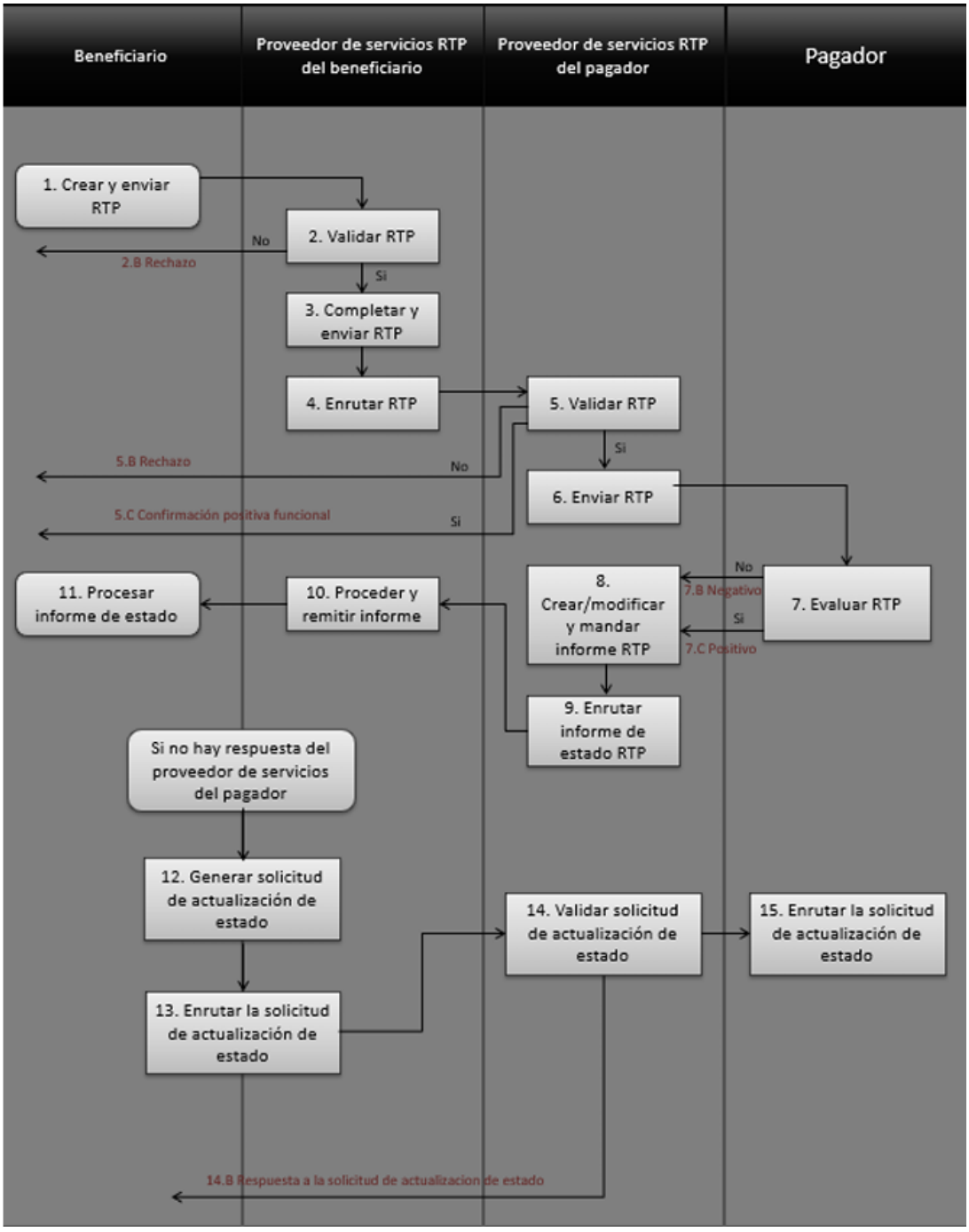
Cuadro 1: Pasos del flujo del esquema SEPA Request-to-Pay (SRTP) y su descripción

Paso	**Descripción**
1. Identificación	Una primera interacción que permite establecer la comunicación entre pagador y beneficiario a través de sus PSP.
2. Envío SRTP al PSP del beneficiario	El beneficiario envía la SRTP a su propio proveedor de servicios. Este mensaje contiene todos los datos esenciales del esquema RTP.
3. Envío SRTP al PSP del pagador	El PSP del beneficiario retransmite la SRTP al PSP del pagador.
4. Presentación SRTP al pagador	La SRTP se muestra al pagador en el canal o dispositivo acordado (aplicación móvil, navegador web, etc.).
5. Informe de estado	La aceptación o el rechazo de la SRTP por parte del pagador se envía de vuelta al beneficiario a través de sus respectivos PSP.

El Operational Scheme Manager (OSM) es la pieza operativa que el EPC ha previsto para ofrecer un servicio centralizado de directorio -el EPC Directory Service (EDS) que facilite la interoperabilidad y el alcance mutuo entre los participantes del esquema SRTP. La OSM actúa como registro autorizado de todos los PSP y demás actores adheridos al esquema y su misión es garantizas que, en todo momento, cualquier PSP pueda localizar el .[en]dpointçorrecto del PSP objetivo, verificar que está homologado y enviar de forma segura los mensajes SRTP. Por tanto, para que un PSP

pase a formar parte activa del esquema SEPA Request-to-Pay deberá en primer lugar superar con éxito las pruebas de conformidad ISO 20022 y obtener la homologación técnica, tras lo cual acceder al portal de la Operational Scheme Manager para registrar su ficha operativa —incluyendo URL de endpoint, roles SRTP, certificados TLS, claves de firma y datos de contacto 24×7— y aceptar los términos de uso del EPC Directory Service, de modo que la OSM valide la disponibilidad y correcta respuesta de sus endpoints y publique la información correspondiente en el EDS público, quedando accesible al resto de participantes; finalmente, el PSP actualiza sus canales de banca electrónica y/o sus APIs de iniciación de pagos para presentar las solicitudes RTP a sus usuarios, garantizar la autenticación fuerte conforme a PSD2 y gestionar de forma integrada los flujos de aceptación, rechazo y cancelación

Flujo básico. El flujo básico de un intercambio RPT es el aiguiente:


Beneficiario
Proveedor de servicios RTP del beneficiario
Proveedor de servicios RTP del pagador
Pagador
1. Crear y enviar RTP
2. Validar RTP
No
2.B Rechazo
Si
3. Completar y enviar RTP
4. Enrutar RTP
5. Validar RTP
Si
5.B Rechazo
No
6. Enviar RTP
5.C Confirmación positiva funcional
Si
No
7.B Negativo
Si
7.C Positivo
7. Evaluar RTP
8. Crear/modificar y mandar informe RTP
9. Enrutar informe de estado RTP
10. Proceder y remitir informe
11. Procesar informe de estado
Si no hay respuesta del proveedor de servicios del pagador
12. Generar solicitud de actualización de estado
13. Enrutar la solicitud de actualización de estado
14. Validar solicitud de actualización de estado
15. Enrutar la solicitud de actualización de estado
14.B Respuesta a la solicitud de actualizacion de estado

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 beneficial. 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.
7B Positivo	Si el pagador decide aceptar el SRTP, se envía una respuesta positiva al proveedor de servicios SRTP por parte del pagador.

(continúa en la siguiente página)

Paso/Función	Descripción
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.).
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.
Fin de la tabla	

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 (Zona Única de Pagos en Euros) 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* (ordenante del pago) y el *beneficiario* (cobrador). 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. En SEPA, existen cámaras de compensación automatizadas (CSM) y redes de pago instantáneo. Por ejemplo, el sistema STEP2 de EBA Clearing procesa transferencias SEPA diarias, mientras que para pagos instantáneos existen redes 24/7 como TIPS (del BCE) o RT1 (de EBA Clearing). 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 ecosistema[13] (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.

[13] `https://redbridgedta.com`

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.

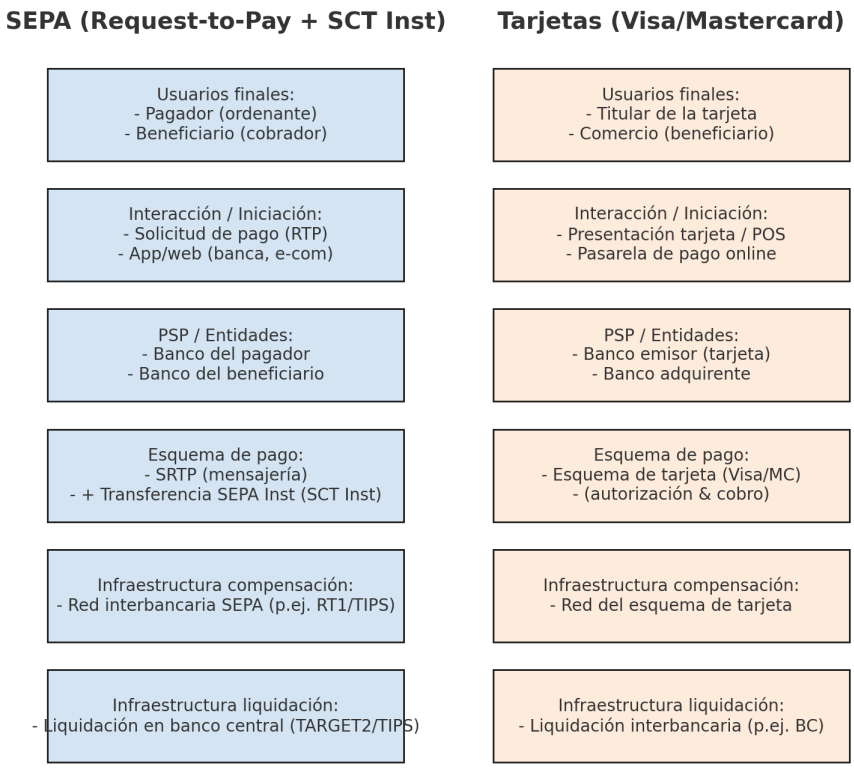


Figura 3: LayerComp

¿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 común[14]. 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 bancario[15]. En resumen, Request to Pay se apoya 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.

[14] `https://cpg.de`, `https://redbridgedta.com`
[15] `https://docs.monei.com`

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]

[16] Transferencia de Estado Representacional

[17] Es 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) ✗ 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.

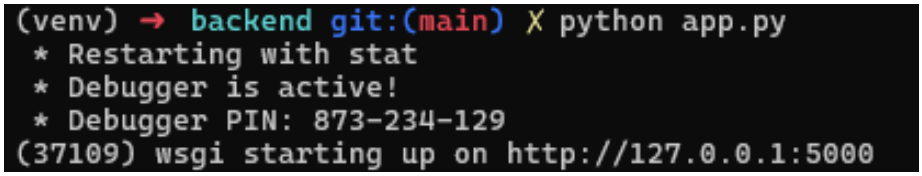


Figura 5: Arranque

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

```
STATIC_FOLDER = os.path.join(os.path.dirname(__file__),
    '../frontend')
app = Flask(__name__, static_folder=STATIC_FOLDER)
app.config.from_object(Config)
db.init_app(app)
# Servir index.html en la ra z
@app.route('/')
def home():
    return app.send_static_file('index.html')

# Servir archivos est ticos desde el directorio frontend
@app.route('/<path:filename>')
def serve_static(filename):
    return send_from_directory(app.static_folder, filename)

# Registrar el blueprint con todos los endpoints de RTP
app.register_blueprint(rtp_blueprint)

@socketio.on('join')
def handle_join(data):
    actor_id = data['actor_id']
    actor = Actor.query.get(actor_id)
    if actor:
        room = f"{actor.role}_{actor_id}"
        join_room(room)
        print(f"Actor {actor_id} se uni  a la sala {room}")

if __name__ == '__main__':
    webbrowser.open("http://127.0.0.1:5000/")
    socketio.run(app, debug=True)
```

- **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.

```
basedir = os.path.abspath(os.path.dirname(__file__))

class Config:
    SQLALCHEMY_DATABASE_URI = 'sqlite:///' +
        os.path.join(basedir, 'rtp.db')
    SQLALCHEMY_TRACK_MODIFICATIONS = False
```

- **models.py** define el modelo de datos que se va utilizar y contiene un ORM[18] 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 $to_dictparaserializarlosdatosaformatoJSONparaenviarlosatravésdeunaAPI.Acontinuacionsemuestraúnicame$

```python
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()

class Actor(db.Model):
    __tablename__ = 'actor'
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(50), unique=True)
    password = db.Column(db.String(50))
    name = db.Column(db.String(50), nullable=False)
    role = db.Column(db.String(20), nullable=False)

    photo_url = db.Column(db.String(200), nullable=True)
        # URL o base64
    iban = db.Column(db.String(34), nullable=True)
    balance = db.Column(db.Float, default=0.0)

    # Campo nuevo: psp_id, que referencia a otro Actor que
        es PSP
    psp_id = db.Column(db.Integer,
        db.ForeignKey('actor.id'), nullable=True)
    # Relaci n para que se pueda acceder con actor.psp
    psp = db.relationship('Actor', remote_side=[id])

    def to_dict(self):
        return {
            "id": self.id,
            "username": self.username,
            "name": self.name,
            "role": self.role,
            "photo_url": self.photo_url,
            "iban": self.iban,
            "balance": self.balance,
            "psp_id": self.psp_id  # Podr amos incluir
                m s info del psp, etc.
        }
```

- **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 çambiar$_e satdo_r tp$", "$rechazar_r tp$"o"$validar_i ban$".

```python
def cambiar_estado_rtp(db, rtp_obj, new_status):
old_status = rtp_obj.status
rtp_obj.status = new_status
db.session.commit()
```

[18] Object-Relational Mapping

```
            # Generar un hash
            hash_input =
                f"{rtp_obj.id}{rtp_obj.iban}{rtp_obj.amount}{old_status}{new_status}".encode('
            hash_value = hashlib.sha256(hash_input).hexdigest()

            nuevo_log = Log(
                rtp_id=rtp_obj.id,
                old_status=old_status,
                new_status=new_status,
                hash_value=hash_value
            )
            db.session.add(nuevo_log)
            db.session.commit()

            return {
                "message": f"RTP {rtp_obj.id} actualizado de {old_status}
                    a {new_status}"
            }

        def rechazar_rtp(db, rtp_obj, motivo):
            return cambiar_estado_rtp(db, rtp_obj, "rechazado")

        def validar_iban(iban, rtp_obj):
            iban = rtp_obj.iban
            if not iban or not isinstance(iban, str) or len(iban) < 15:
                return cambiar_estado_rtp(db, rtp_obj, "rechazado")
            if not (iban[:2].isalpha() and iban[2:].replace(' ',
                '').isalnum()):
                return cambiar_estado_rtp(db, rtp_obj, "rechazado")
            return {
                "message": f"RTP {rtp_obj.id} validado con IBAN correcto"
            }
```

- **utils roles.py** contiene el decorador rolerequired 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.

```
            def role_required(required_role):
            """
            Decorador que verifica que el actor que invoca el endpoint
            tenga el rol adecuado.
            Se espera que en el body JSON llegue "actor_id".
            """
            def decorator(f):
                @wraps(f)
                def wrapper(*args, **kwargs):
                    data = request.get_json() or {}
                    actor_id = data.get('actor_id')
                    if not actor_id:
                        return jsonify({"error": "Se requiere actor_id
                            para esta acci n"}), 400

                    actor = Actor.query.get(actor_id)
                    if not actor:
                        return jsonify({"error": "Actor no encontrado"}),
                            404
```

```
                if actor.role != required_role:
                    return jsonify({"error": f"Rol '{actor.role}' no
                        tiene acceso a esta acci n"}), 403

                # Si el rol coincide, continuamos
                return f(*args, **kwargs)
            return wrapper
        return decorator
```

- **services.py** contiene la capa de negocio. Cada función del fichero orquesta la consulta/actualizacion 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 servcies.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.

```
rtp_blueprint = Blueprint('rtp', __name__)
```

```
@rtp_blueprint.route('/login', methods=['POST'])
def login():
    data = request.get_json()
    username = data.get('username')
    password = data.get('password')
    if not username or not password:
        return jsonify({"error": "Faltan credenciales"}), 400

    actor = Actor.query.filter_by(username=username).first()
    if not actor:
        return jsonify({"error": "Usuario no existe"}), 404

    if actor.password != password:
        return jsonify({"error": "Contrase a incorrecta"}),
            401

    # Login exitoso
    return jsonify({
        "message": "Login correcto",
        "actor_id": actor.id,
        "role": actor.role,
        "name": actor.name
    })
```

```
@rtp_blueprint.route('/profile', methods=['POST'])
@role_required('payer')
def update_profile():
    data = request.get_json() or {}
```

```
                        actor_id = data.get('actor_id')
                        actor = Actor.query.get(actor_id)
                        if not actor:
                            return jsonify({"error": "Actor no encontrado"}),
                                404

                        # Campos opcionales
                        new_photo = data.get('photo_url')
                        new_iban = data.get('iban')
                        new_balance = data.get('balance')

                        if new_photo is not None:
                            actor.photo_url = new_photo
                        if new_iban is not None:
                            actor.iban = new_iban
                        if new_balance is not None:
                            actor.balance = float(new_balance)

                        db.session.commit()
                        return jsonify({"message": "Perfil actualizado",
                            "actor": actor.to_dict()})
```

- **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.

```
actors_blueprint = Blueprint('actors', __name__)

@actors_blueprint.route('/actors', methods=['POST'])
def create_actor():
    data = request.get_json()
    name = data.get('name')
    role = data.get('role')  # 'beneficiary',
        'psp_beneficiary', 'psp_payer', 'payer'

    if not name or not role:
        return jsonify({"error": "Faltan campos requeridos:
            name, role"}), 400

    # Validar que el role sea uno de los cuatro permitidos:
    valid_roles = ['beneficiary', 'psp_beneficiary',
        'psp_payer', 'payer']
    if role not in valid_roles:
        return jsonify({"error": f"Rol '{role}' no v lido"}),
            400

    actor = Actor(name=name, role=role)
    db.session.add(actor)
    db.session.commit()

    return jsonify({"message": "Actor creado", "id": actor.id,
        "role": actor.role}), 201
```

- **ext socketio.py** simplemente es un fichero de conveniencia que crea la instancia soccketIO. 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.

```
from flask_socketio import SocketIO, join_room, emit

socketio = 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 Ïnicio", 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 Çrear 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 logsrealiza una solicitud `GET /logs`, recupera el historial de transiciones 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.

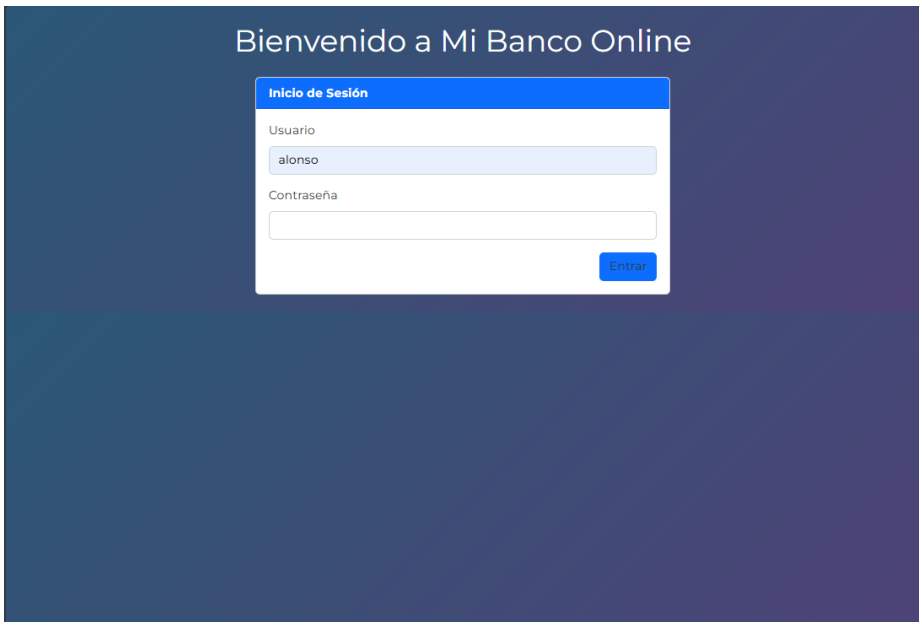


Figura 6: LoginForm

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.

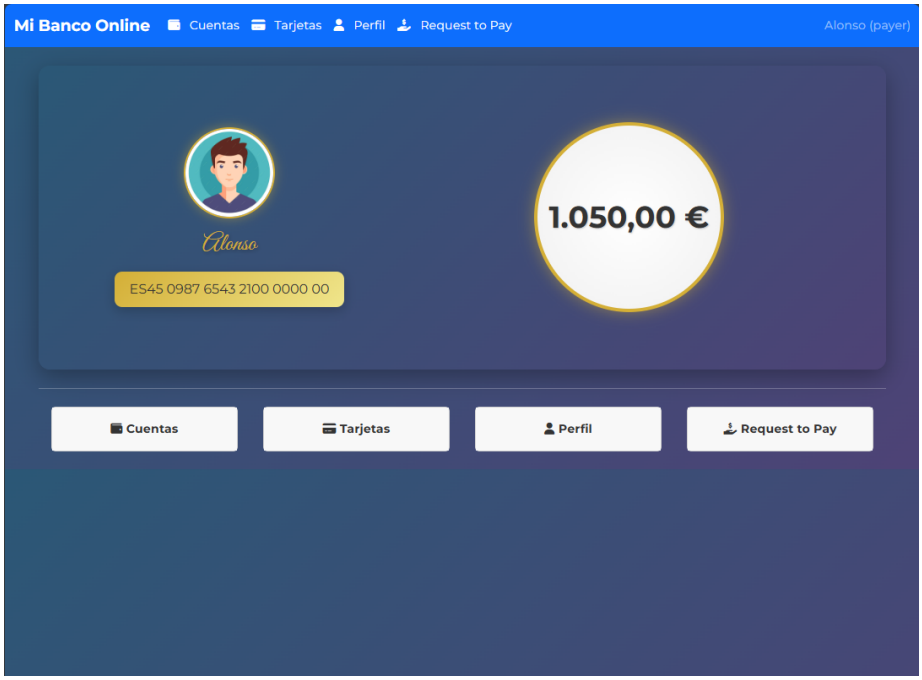


Figura 7: DashBoard

Lo verdaderammente novedoso del prototipo, y por tanto, el *producto* que se "vende.es la nueva opcion *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:

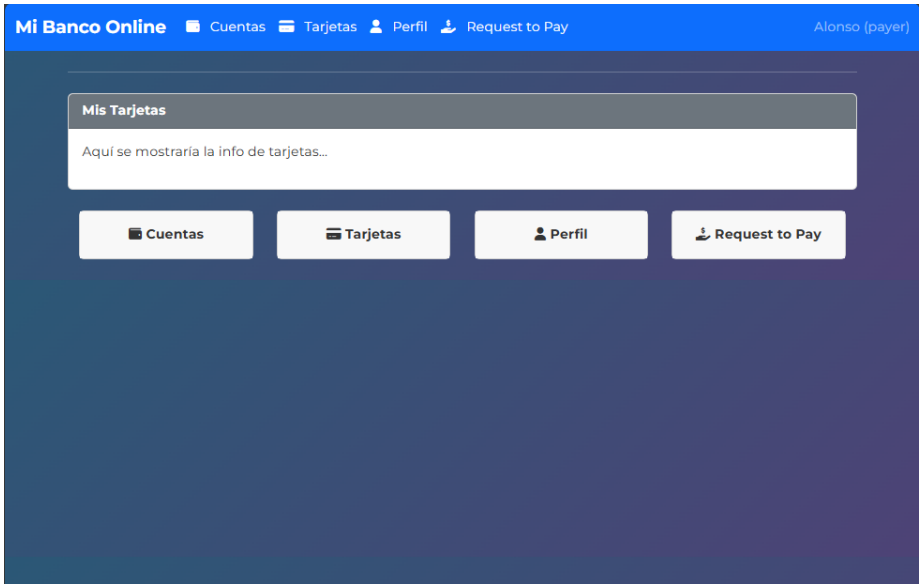


Figura 8: Tarjetas

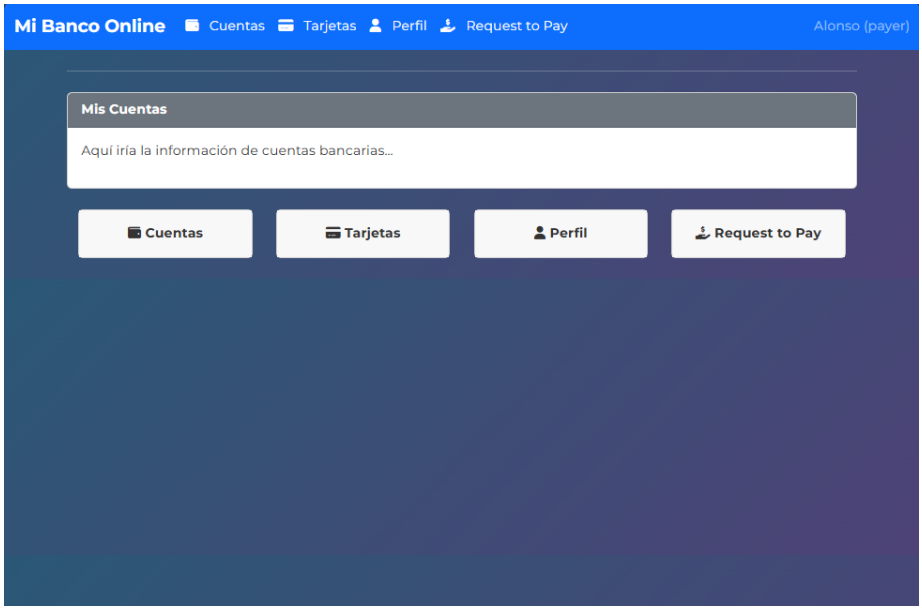


Figura 9: Cuentas

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

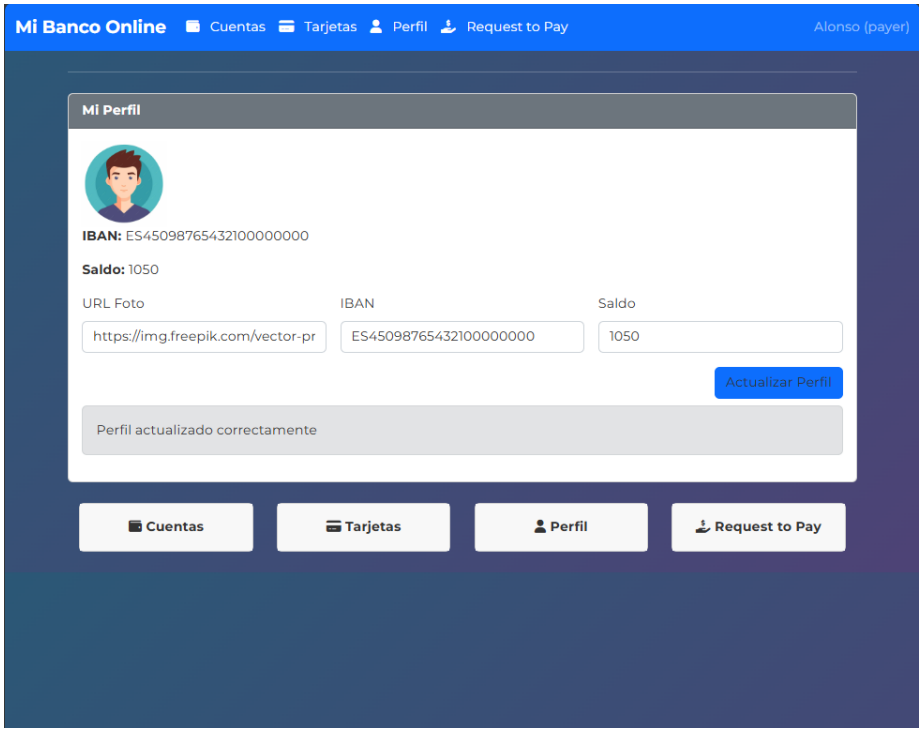


Figura 10: Perfil

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.

```python
@rtp_blueprint.route('/rtp', methods=['POST'])
@role_required('beneficiary')
def crear_rtp():
    data = request.get_json()
    # ID del beneficiario que est  logueado
    beneficiary_id = data.get('actor_id')
    # IBAN del pagador
    payer_iban = data.get('payer_iban')
    # Importe solicitado
    amount = data.get('amount')
```

```
        if not payer_iban:
            return jsonify({"error": "Falta iban"}), 400

        if not amount:
            return jsonify({"error": "Falta amount"}), 400

        # Delegaci n a la capa de negocio
        result = crear_rtp_service(data)
        status = 201 if "error" not in result else 400
        return jsonify(result), status
```

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 validación.
4. Con los datos verificados, la lógica de negocio toma el relevo y, al finalizar, la ruta convierte su resultado en una respuesta HTTP adecuada (éxito o error).

```
def crear_rtp_service(data):
    beneficiary_id = data.get('actor_id')
    payer_iban      = data.get('payer_iban')
    amount          = data.get('amount')

    # 1) Validar beneficiario y su PSP
    beneficiary = Actor.query.get(beneficiary_id)
    if not beneficiary or beneficiary.role != 'beneficiary':
        return {"error": "El actor no es beneficiario o no existe"}

    if not beneficiary.psp_id:
        return {"error": "El beneficiario no tiene PSP asociado"}
    psp_benef_id = beneficiary.psp_id

    # 2) Localizar pagador y su PSP
    payer = Actor.query.filter_by(iban=payer_iban,
        role='payer').first()
    if not payer:
        return {"error": "No se encontr  un pagador con ese IBAN"}

    if not payer.psp_id:
        return {"error": "El pagador no tiene PSP asociado"}
    psp_payer_id = payer.psp_id

    # 3) Crear y guardar el RTP
    nuevo_rtp = RTP(
        iban=payer_iban,
        amount=amount,
        beneficiary_id=beneficiary.id,
        psp_beneficiary_id=psp_benef_id,
        psp_payer_id=psp_payer_id,
        payer_id=payer.id
    )
```

```
                db.session.add(nuevo_rtp)
                db.session.commit()

                # 4) Notificaci n en tiempo real al PSP del beneficiario
                socketio.emit(
                    'rtp_created',
                    nuevo_rtp.to_dict(),
                    room=f'psp_beneficiary_{psp_benef_id}'
                )

                return {
                    "message": "RTP creado correctamente",
                    "id": nuevo_rtp.id
                }
```

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.

```
            @rtp_blueprint.route('/rtp/<int:rtp_id>/validate-beneficiary',
                methods=['POST'])
            @role_required('psp_beneficiary')
            def validar_beneficiario(rtp_id):
                result = validar_beneficiario_service(rtp_id)
                status = 200 if "error" not in result else 400
                return jsonify(result), status

            @rtp_blueprint.route('/rtp/<int:rtp_id>/route', methods=['POST'])
            @role_required('psp_beneficiary')
            def enrutar_rtp(rtp_id):
                result = enrutar_rtp_service(rtp_id)
                status = 200 if "error" not in result else 400
                return jsonify(result), status
```

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.

```
def validar_beneficiario_service(rtp_id):
    rtp_obj = RTP.query.get(rtp_id)
    if not rtp_obj:
        return {"error": "RTP no encontrado"}
    result = cambiar_estado_rtp(db, rtp_obj,
        "validado-beneficiario")

    socketio.emit(
        'rtp_validated_beneficiary',
        rtp_obj.to_dict(),
        room=f'psp_beneficiary_{rtp_obj.psp_beneficiary_id}'
    )
    return result

def enrutar_rtp_service(rtp_id):
    rtp_obj = RTP.query.get(rtp_id)
    if not rtp_obj:
        return {"error": "RTP no encontrado"}
    result = cambiar_estado_rtp(db, rtp_obj, "enrutado")

    socketio.emit(
        'rtp_routed',
        rtp_obj.to_dict(),
        room=f'psp_payer_{rtp_obj.psp_payer_id}'
    )
    return result
```

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.

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

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).

```
def validar_payer_service(rtp_id):
    """
    Validaci n por el PSP del pagador.
    1) Si el pagador no dispone de fondos suficientes, se fuerza
        el rechazo.
    2) En caso contrario, se marca el RTP como validado por el PSP
        del pagador.
    """
    rtp_obj = RTP.query.get(rtp_id)
    if not rtp_obj:
        return {"error": "RTP no encontrado"}

    # Buscar al pagador asociado y verificar su saldo
    from models import Actor
    payer_actor = Actor.query.get(rtp_obj.payer_id)
    if not payer_actor:
        return {"error": "El payer asignado a este RTP no existe"}

    if payer_actor.balance < rtp_obj.amount:
        # Rechazo inmediato por falta de fondos
        return rechazar_rtp(db, rtp_obj, "Saldo insuficiente (PSP
            forz  cancelaci n)")

    # Fondos suficientes     validaci n satisfactoria
    result = cambiar_estado_rtp(db, rtp_obj, "validado_payer")

    # Notificaci n al pagador (front-end) para que tome su
        decisi n final
```

```
                socketio.emit('rtp_validated_payer',
                              rtp_obj.to_dict(),
                              room=f'payer_{rtp_obj.payer_id}')

                return result
```

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 aplicación cliente muestre la notificación y solicite al usuario la decisión definitiva (aceptar o rechazar el cobro).

Con estas dos piezas de código 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.

- **Decisión 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 último paso del flujo se implementa, como los anteriores, con una ruta HTTP y una pieza de lógica de negocio.

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

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.

```
def decision_payer_service(rtp_id, data):
    """
    Decisi n final del pagador: 'aceptado' o 'rechazado'.
    - Si es 'aceptado', se descuentan los fondos.
    - Se registra el nuevo estado y se notifica al beneficiario.
    """
```

```
                rtp_obj = RTP.query.get(rtp_id)
                if not rtp_obj:
                    return {"error": "RTP no encontrado"}

                decision = data.get('decision')
                if decision not in ["aceptado", "rechazado"]:
                    return {"error": "Decisi n no v lida"}

                # Descontar fondos si procede
                if decision == "aceptado":
                    from models import Actor
                    payer_actor = Actor.query.get(rtp_obj.payer_id)
                    if not payer_actor:
                        return {"error": "El payer asignado no existe"}
                    if payer_actor.balance < rtp_obj.amount:
                        return rechazar_rtp(db, rtp_obj,
                                            "Saldo insuficiente en el  ltimo
                                               momento")
                    payer_actor.balance -= rtp_obj.amount
                    db.session.commit()

                result = cambiar_estado_rtp(db, rtp_obj, decision)

                # Aviso en tiempo real al beneficiario
                socketio.emit('rtp_decision',
                              rtp_obj.to_dict(),
                              room=f'beneficiary_{rtp_obj.beneficiary_id}')

                return result
```

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 elección recibida sea «aceptado» o «rechazado». Cualquier otro valor provoca un error de validación.
3. Si la opción es «aceptado», se confirma que el pagador dispone de saldo suficiente y, de ser así, 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`.

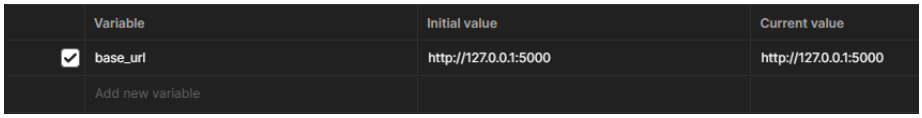


Figura 11: Variable `baseURL` en Postman

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.

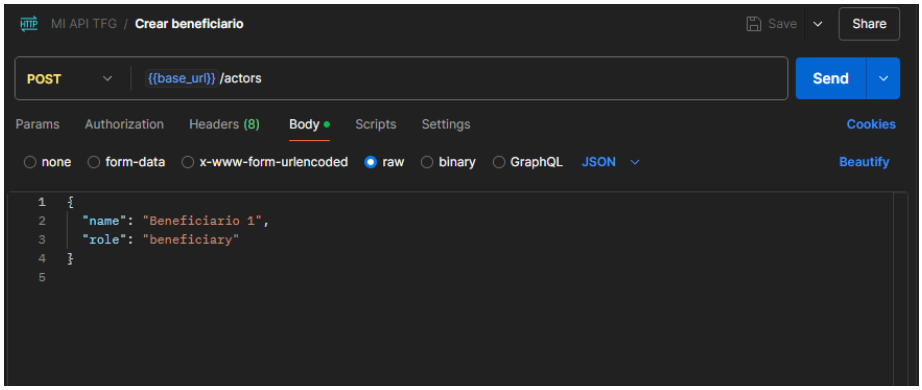


Figura 12: crearActor

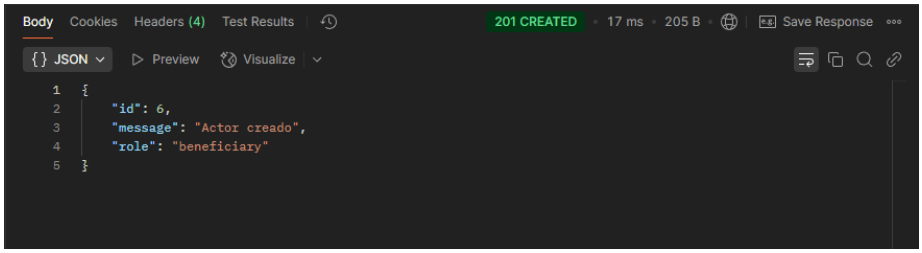


Figura 13: crearActorResponse

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

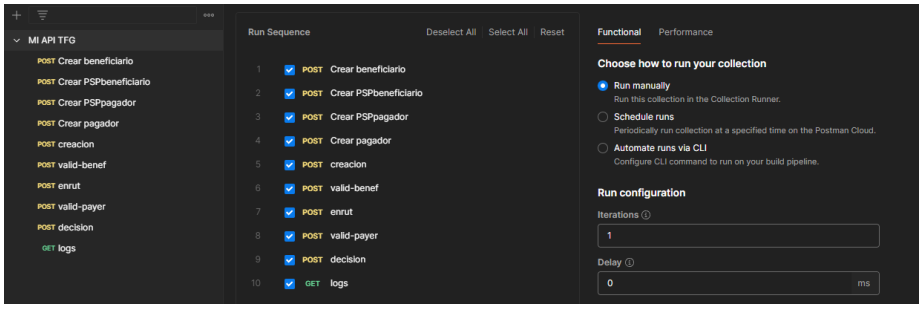


Figura 14: MiAPITFG

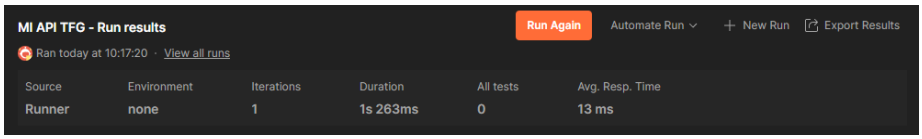


Figura 15: RunResults

Las peticiones relevantes del flujo son:
numerate

Creación RTP

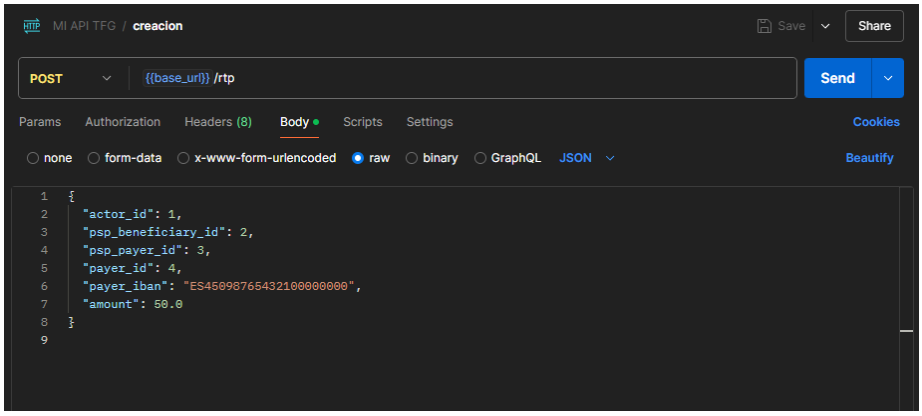


Figura 16: crearRTP

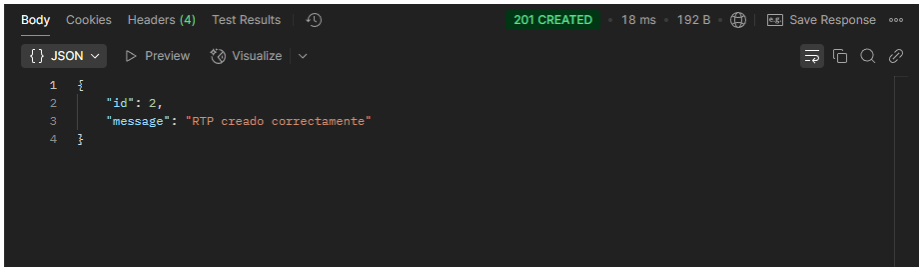


Figura 17: crearRTPResponse

Validación y enrutado del PSP del beneficiario

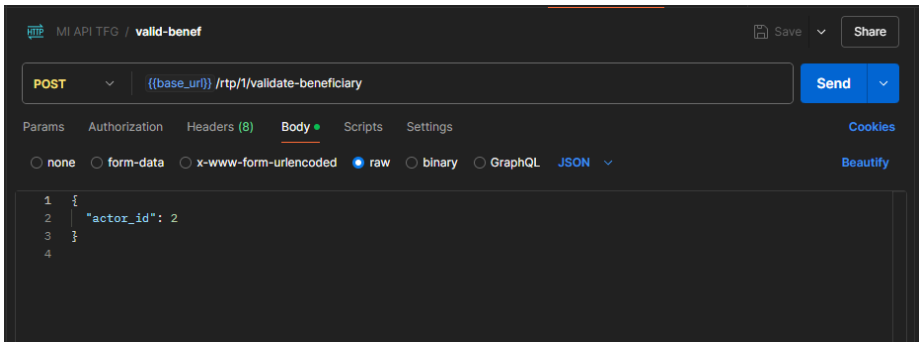


Figura 18: validarBenef

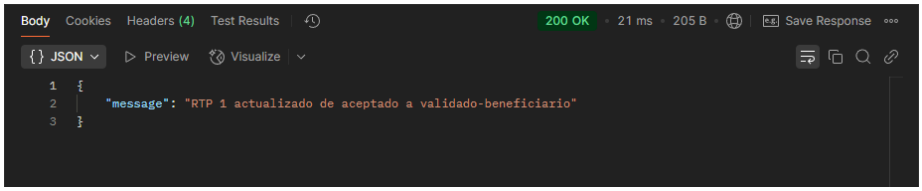


Figura 19: validarBenefResponse

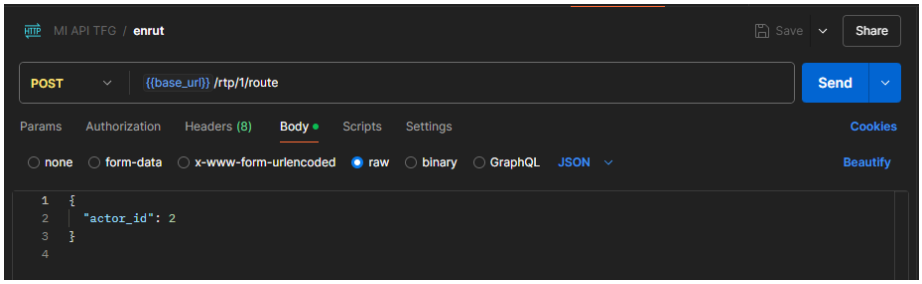


Figura 20: enrutarBenef

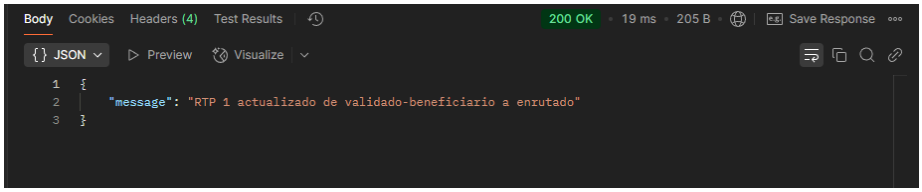


Figura 21: enrutarBenefResponse

Validación del PSP del pagador

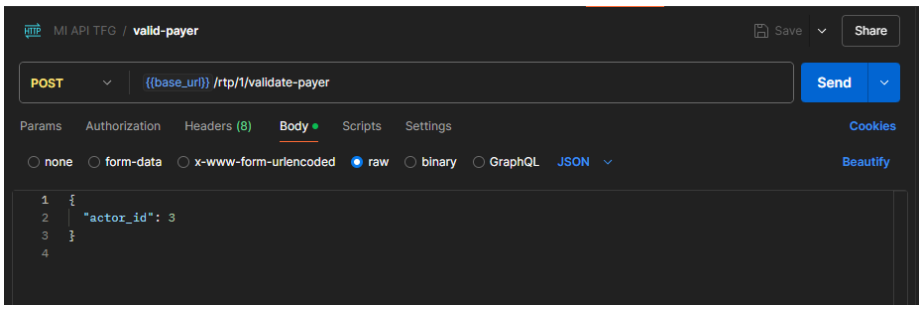


Figura 22: validarPayer

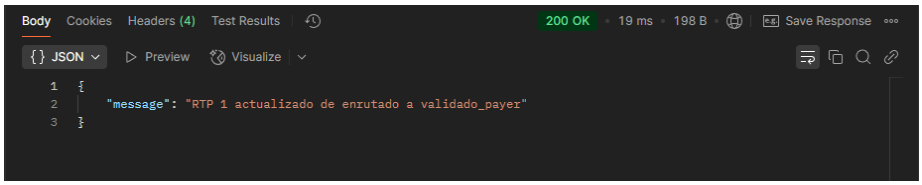


Figura 23: validarPayerResponse

Decisión final

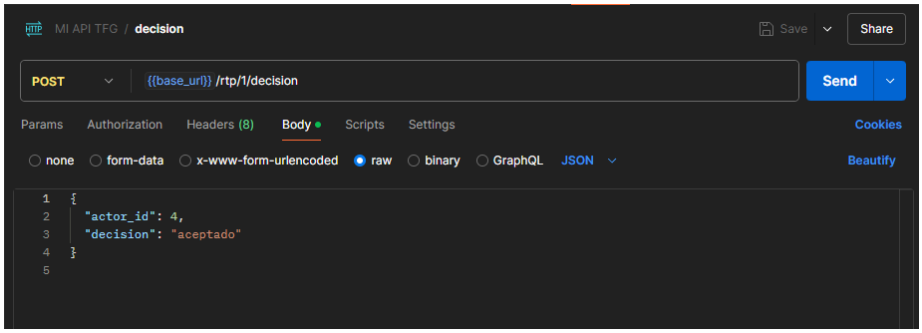


Figura 24: decision

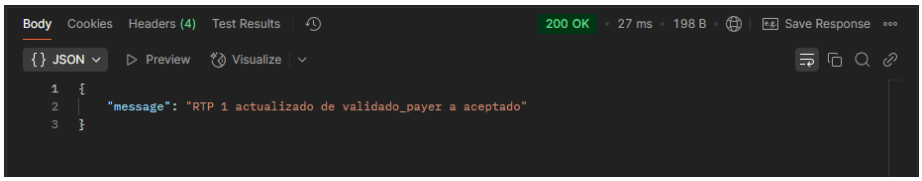


Figura 25: decisionResponse

A Códigos

```
fID1 = fopen('myoutput1.txt','r');
for n = 1:4
    b = fscanf(fID1,'%7u %7u %7u \r',3);
    btotal = b(1)+b(2)+b(3);
    fprintf('%7u + %7u + %7u = %7u \r', b(1), b(2), b(3), btotal)
end
```

Listing 1: Código de ejemplo.

Referencias

Índice de Figuras

Índice de Tablas

Índice de Códigos