payments strategy forum

Dec 2017

Collaborative Requirements and Rules for the End-User Needs Solutions

Request to Pay Technical Solution Blueprint

Project/Programme Manager:	Ivan Litovski
Sponsor:	Payments Strategy Forum
Date of Final Approval:	30 11 2017
Approved by:	Sian Williams

Version / Document History

Version No	Date	Author	Comments
0.9	24/11/2017	Ivan Litovski Mike Banyard Simon Brooks Adrian Burholt Duncan Ng'enda	Draft solution Blueprint focussing on Request to Pay APIs

This is not the final version of the API specification for Request to Pay.

The NPSO will carry on into the next year to further refine this in line with the Indicative Request to Pay plan published in the End-user Needs Requirements and Rules blueprint.

Contents

1 Introduction 5
1.1 Purpose of this Document 5
2 Quick Overview of Request to Pay 7
3 Architecture Principles 8
3.1 Optimisation Criteria 8
3.2 Decoupled Systems Based on Web Standards 8
3.3 Functionality Encapsulation 8
3.4 Traffic Pattern Optimisation 8
3.5 Automation, Managed Tasks 9
3.6 Service Instrumentation 9
3.7 Support Cloud Portability 9
3.8 Mobile First, API First 9
4 Open API High Level Approach 10
4.1 What are APIs? 10
4.2 API Ecosystem 10
4.2.1 Validation of participant implementations 11
4.3 API Versioning 12
4.3.1 Operating multiple API versions concurrently 12
4.3.2 Deprecation of previous versions 12
4.3.3 Retirement of previous versions 12
4.4 API Security 12
5 Security Approach 14
5.1 Trust Management 14
5.2 Securing Data in Transit 14
5.3 Securing Data at Rest 14
5.4 Message Authentication 14
5.5 Common Attack Vectors and Mitigation 15
5.5.1 Biller impersonation 15
5.5.2 TPP impersonation 15
5.5.3 Message tampering and MITM 15
6 Functional Description 16
6.1 Overview 16
6.2 Third-party Provider Identity and Onboarding 16
6.3 End-user Identity 17
6.4 End-user Onboarding 19
6.4.1 Onboarding based on existing relationship 19
6.4.2 Onboarding to a new relationship 19
6.4.3 Onboarding via invitation from Payee 19

6.5 State Management 20
6.6 Resilience to Failures 21
6.6.1 Out of sync repositories 21
6.6.2 Repository operator ceases operation 21
6.6.3 App operator ceases operation 22
6.6.4 Index failures 22
6.6.5 Rejection failures 22
6.7 Executing a Payment 22
6.7.1 Non-PSD2 Electronic Payment 22
6.7.2 PSD2 electronic Payment 23
6.7.3 Cash payment 24
6.8 Request to Pay walkthrough 25
6.9 Functional requirement summary 26
7 Architecture Blueprint 27
7.1 Overview 27
7.2 Message Structure 28
7.3 Message Flows & Sequence Diagrams 29
7.3.1 Registering a new user 29
7.3.2 Message exchanges 30
7.3.3 Delivery failures and state synchronisation 31
7.4 Components 31
7.4.1 Index 31
7.4.2 Message Repository 36
7.4.3 Applications 43
8 Request to Pay Demonstrator 45
9 API Design Guidelines 46
9.1 Overview 46
9.2 RESTful API Design Guidelines 46
9.2.1 API URL Naming Conventions 46
9.2.2 URI Versioning 46
9.2.3 URI Format 46
9.2.4 Resource Naming Convention 47
9.2.5 Modelling Resources and Sub-Resources 47
9.2.6 HTTP Verbs 48
9.2.7 API Payload Format 48
9.2.8 API Headers 48
9.3 Exception Handling 49
9.3.1 Error Handling 49

1 Introduction

For the majority of people, the technical aspects of payments are invisible. They run in the background supporting various activities in our lives that require the movement of money. Examples include receiving an income, paying bills, making a mortgage or rent payment, or buying groceries. The way we make payments and interact with payment systems has changed dramatically in the last few years. We identified these changes in the Strategy and acknowledge that a growing number of end-users' needs are not completely met by the current payment systems.

A predominant theme was the need for end-users to have:

- More control over their payments.
- More flexibility over how much, when, and how they pay.
- Increased transparency in their interactions with payments.

There is broad consensus that a Request to Pay service will help address the detriments mentioned above and bridge the growing needs gap. We designed a Request to Pay service that specifically addresses these detriments.

What is Request to Pay?

Request to Pay is a communication mechanism that will allow a payee (government, businesses, charities and consumers) to send a message to a payer requesting a payment. Through Request to Pay, a payee will be able to notify a payer of a payment that requires their attention and in return, the payer will be able to respond to the payee. For example, the payer will be able to accept the request and make full or partial payments; decline it; request an extension of the time period in which they can make the payment; or request more information.

When a payer accepts the request, they will be able to pay using a choice of available methods, and the acceptance will automatically trigger the payment being made. End-users (individuals, SMEs, corporates and government) could benefit from Request to Pay. Payees will be provided with visibility on what the payer's intention is with regards to a bill payment. Currently, once a payee sends out a bill, they have limited visibility on whether the payer will make a payment or not and when they will pay.

Increased visibility has a positive impact on cash flow management, payment reconciliation, debt management and overall customer relationship management. Cash flow management is especially important to SMEs who tend to have limited cash reserves making them vulnerable to cash flow challenges. This benefit is dependent on the payer choosing to respond to a Request to Pay. Request to Pay provides visibility to the payer on outgoing payments; it opens a communication channel to the payee; and it provides a tool through which a payer can flex how they make their payments - when, how, and how much.

Request to Pay is independent of the payment mechanism used to make a payment. We have taken an approach to separate the messaging and the payment mechanism in our design. This approach provides more flexibility to both payers and payees on the payment mechanisms through which they make and collect payments, as well as fostering competition for both the messaging component and the payment mechanism of Request to Pay, which could be provided by different service providers.

1.1 Purpose of this Document

Three End-User Needs (EUN) solutions were prioritised in the Strategy: 'Request to Pay', 'Assurance Data' and 'Enhanced Data'. The PSF has developed a minimum set of requirements and rules that any provider of these solutions would have to meet in order to offer them to users.

This document captures:

- High level solution principles
- Functional service description

- Solution blueprint

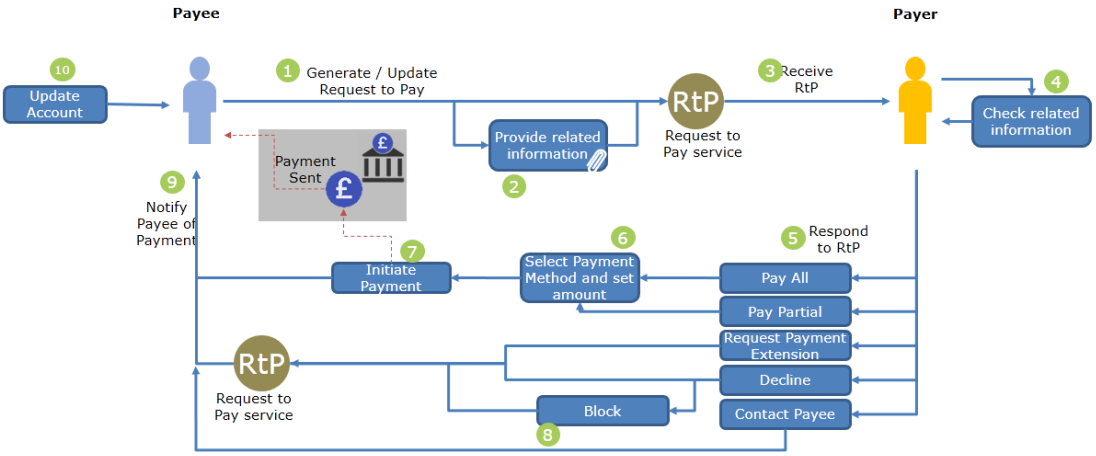
This architecture blueprint document will be used to:

- Describe agreed functional characteristics of the service
- Guide detailed service design and architecture

The specifications detailed in this document will form part of the suite of common standards defined by the Payments strategy forum. Once finalised, they will be handed over to the NPSO, who will be responsible for administering them. The requirements will serve as a core standard on which the competitive market can build rich and compelling propositions for the benefit of end-users.

2 Quick Overview of Request to Pay

Request to Pay enables businesses and consumers to send and receive requests for payment, providing a range of options for data provided in the request, and a range of options a payer has at their disposal to respond to a particular request. The service has been described via separate business use cases and requirements documents, however a summary of use cases is presented in Figure 1 below.



1. A payee generates a new Request to Pay (or updates an existing RtP), which is then sent to the payer.
2. A payee has the option to provide additional information to the payer, which could take the form of a hyperlink to related information stored elsewhere or an attached document.
3. The payer receives the RtP.
4. The payer reviews related information associated to the received request – this could be a hyperlink or an attached file.
5. The payer responds to the RtP, at which point they have a list of options for payment; pay all, pay partial, request payment extension, decline or contact payee.
6. The payer selects the payment method they want to respond to the payment request – these will be a list of methods that the Payee accepts. The payer then can set the amount that they want to pay for a single instalment.
7. The payer initiates a payment to respond to payee's request.
8. A payer can block a payee from sending them requests. The payee will be notified, and any future requests will not be received by the payer (unless they choose to unblock the payee).
9. The payee receives a notification with the payer's response
10. The payee updates payer's billing account based on the information that has been received.

Figure 1 Request to Pay end-to-end journey

3 Architecture Principles

This section describes key architectural principles for Request to Pay implementation.

3.1 Optimisation Criteria

Key optimisation criteria for the Request to Pay architecture is service Reliability. Service architecture must provide assurance that messages are delivered and functional recovery mechanisms in case of service failure by either central infrastructure or individual participant infrastructure, or data loss within any participant (Third Party) data store and multiple participant data stores. Such events should be treated as Business-As-Usual events rather than exception / disaster recovery events. Dealing with these events should be automated and included in every implementation.

Secondary optimisation criteria are agility and extensibility.

Agility is primarily to ensure the service can evolve gracefully and in a controlled manner – through, for example, versioning of service specification and implementations, ensuring future versions of the software can seamlessly communicate with previous versions for interoperability.

Extensibility in the design can ensure modular implementations and add-on services to be developed without impacting core functionality. For example, a third party may develop server-side message agents or message processors to support intelligent functions within their systems of engagement (e.g. apps).

3.2 Decoupled Systems Based on Web Standards

Decoupling Systems of Engagement – with the consumers, Third Party Providers (TPPs) and central infrastructure operators from service systems and systems of record can ensure that a variety of systems can interact over clearly defined boundaries – APIs. This ensures that various service components can be independently developed with technologies appropriate to the problem space, while ensuring interoperability.

For example, a consumer may want to use a native iOS or Android application, while a large corporate biller may want to use a batch process with a web interface to interact with the service. Similarly, repository implementers may choose a variety of technologies to implement e.g. repositories.

Use of web standards for interoperability between participants is essential. Use of modern concepts such as API Management, RESTful services, JSON data payloads, standard HTTP based interactions and delegated authority via oAuth2 where applicable would ensure ease of integration and broadest possible adoption.

3.3 Functionality Encapsulation

Each system component will have a clear, definite set of functions required for operating within the system. For example, repository functions are to be clearly defined and limited to what is required from a repository. This does not prevent implementers from providing additional functions or producing an implementation that may perform as both a repository and a system of engagement, if standard functions of both are independently compliant.

3.4 Traffic Pattern Optimisation

The message flows in the architecture are to be optimised to ensure manageable volumes and controllable traffic patterns. Primary goal here is to limit the number of API calls per transported message,

using techniques such as caching. Where possible, messages may be queued for asynchronous processing to ensure service availability.

3.5 Automation, Managed Tasks

This principle refers to automation of human interactions required to operate the service, including onboarding third parties, third-party self-service functions and similar business processes. These functions should be performed through e.g. a web user interface to enforce process requirements, reduce the amount of effort and manual intervention required to operate the service.

3.6 Service Instrumentation

In order to operate, monitor and continuously improve the service, a level of instrumentation is necessary in both central infrastructure and TPP provided infrastructure. Solution architecture – as well as terms of service for TPPs – should provide for regulation compliant Quality-of-Service / aggregated data collection.

3.7 Support Cloud Portability

While there is no expectation of implementing any particular part of the system on the Cloud, using modern cloud technologies such as containers and container management is clearly desirable, even in private datacentre or on premise implementations. This can help ensure availability (e.g. multi-data centre, multi-provider or multi-cloud implementation), and optimise costs by scaling service to demand.

3.8 Mobile First, API First

In developing the Request to Pay service, the implementers are required to adopt outside-in, API-First approach. The basis of the API-First approach is to define interfaces first, then rapidly test and adapt them based on a variety of client use cases, all before building actual services.

Application and user interface providers should use mobile-first approach, building attractive user experiences.

4 Open API High Level Approach

4.1 What are APIs?

APIs allow controlled interchange of data between disparate systems based on open standards and modern web technologies. Seen as an evolution of Service Oriented Architecture (SOA), APIs take concepts of web-services from enterprise to internet-scale.

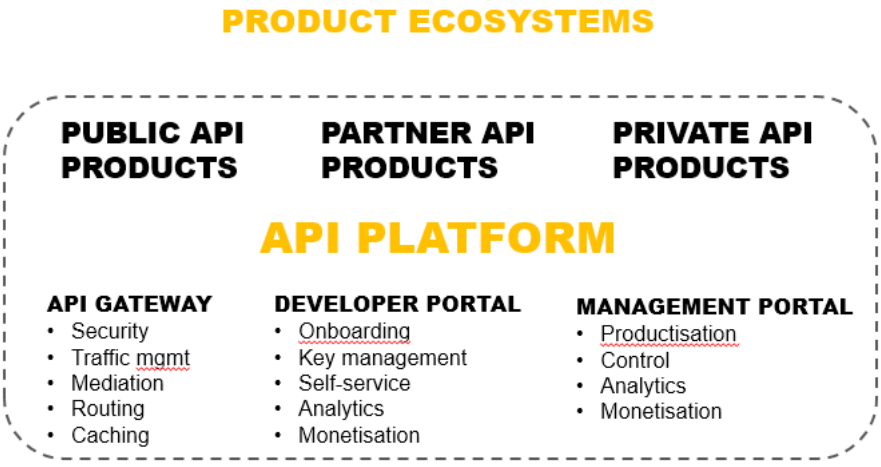


Figure 2 API Platform and key functions

Using APIs, functionality and data from traditional product platforms can be securely exposed to a variety of consumers in an internet-wide product ecosystem.

API Platform plays a crucial role in exposing APIs. The three key functions of an API Platform being:

- API Gateway – an "HTTP firewall" controlling access and performing real-time transformations, traffic management, caching and a variety of other functions.
- Developer Portal – a web based application providing developer experience – from onboarding, API catalogue, managing subscriptions to "API products" to billing for monetised APIs.
- Management Portal – a web based application providing a control interface for the API gateway. Allows publishing APIs, creating API products with pricing, analytics etc.

A variety of vendors provide API platforms with varying feature sets and product focus.

For clarity, the terminology adopted in this document is as follows:

- API Definition – An Open API Specification 2.0 compliant definition [Ref 2]
- API Operation – a RESTful HTTP service exposed for use over the open internet [Ref 3]
- API – a set of API Operations grouped for functional completeness
- API Management platform – a product providing functionality to create a publish APIs, provide an API catalogue, developer onboarding and other services

4.2 API Ecosystem

Request to Pay implementation may include a layered architecture comprising central service, and different types of Third-Party participants (for example, message repository operator and end-user application operators).

Figure 3 A view of the Request to Pay participant Ecosystem

Of crucial importance for the development of the ecosystem are Repository operators. While the core APIs will be standardised, Repository operators will be able to greatly expand on the default APIs and offer extended API services. Through this, Repository operators would compete for innovators building consumer applications, as well as a broad range of corporate and business applications that may need integration.

4.2.1 Validation of participant implementations

Participants may produce various implementations of message repositories or end-user applications, and associated APIs. In order to be accredited for operating Request to Pay services, participant implementation must pass a Compatibility Test Suite – a set of tests to establish that particular implementation complies with expected functionality.

API Definition: All APIs in use by Request to Pay system are to be centrally defined by the NPSO. The specification is to be created in Open API Specification 2.0 (OAS2) compliant format, and in accordance with guidelines in API Design Guidelines. .

API Definitions shall include:

- Standardised URI path for each API operation.
- URI Path shall include API version number
- All request parameters
- For GET and DELETE requests, this shall include all parameters in the request URI
- For POST & PUT requests, this shall include definition of submitted content, including any structured data definitions (JSON schemas).
- All non-standard headers
- Use of standard headers shall be documented only where it may modify behaviour of any part of the implementation – e.g. caching etc.
- All HTTP response codes generated by the API
- Where possible, response codes expected to be generated by the execution environment
- Where response carries a usable payload – e.g. GET request, some POST/PUT requests, the response content is to be documented, including any data / JSON schemas for composite JSON objects.

Preferred data format for Request to Pay REST transactions is JSON (ECMA-404), and schemes are to be provided in JSON-schema (ietf-draft-6).

4.3 API Versioning

New API versions may, from time to time, be defined by the Request to Pay scheme. Version must be clearly visible in the request path portion of the API URL.

4.3.1 Operating multiple API versions concurrently

If multiple versions of an API are defined by the Request to Pay scheme, participants may operate multiple versions of the APIs.

The caller (participant issuing a HTTP request), must initially request the most recent version of the API it is certified for.

If a target participant does not implement the version requested, the caller must fall back to previous version of the API with the appropriate version number in the URL, retrying until an endpoint is reached that the target implements.

Similarly, if a target participant supports newer version than that requested, they must respond in the format corresponding to the format requested / version specified in the URL path.

4.3.2 Deprecation of previous versions

Deprecation of an API means that there is a newer, preferred version of the API at an appropriately formed URI path. However, the deprecated version remains a valid API endpoint for all practical purposes until retired.

Deprecation can only be initiated centrally, by the NPSO. Deprecated API version implementation should operate without change, with addition of deprecation notice in the response meta-data (e.g. HTTP header). Deprecation notice may include a web link explaining deprecation warning, the URI of the latest version or, in case deprecated version is due to be retired, the date and time of retirement.

4.3.3 Retirement of previous versions

Retirement of an API version would require changes to the software used by all participants and is therefore not a desirable outcome. Due to the possible impact on the service, retirement of APIs must be a carefully defined and managed process.

There are several distinct cases for retiring an API:

1. All registered participants already use and implement newer API version – retirement is recommended, notification to all service participants is necessary.
2. A newer version of the API is available and is preferred, but not all participants implement it. – Retirement is not recommended, see previous section on running multiple API versions concurrently.
3. A security flaw is discovered in an API definition, however no newer version is available. This must be a carefully managed process depending on how critical the API is for the operation of the service.

4.4 API Security

In API security, it may be necessary to establish:

4. Caller identity (e.g. identity of the participant invoking the API)
5. Responder identity (e.g. identity of the participant responding to the API call)
6. Identity of an end user on behalf of whom the call is made.
7. Authorisation of the third party provider to act on behalf of a particular user

The approach for establishing caller and responder identity (cases 1 & 2 above), is described in section 5.2. Simply put, in TLS connection setup, parties must present certificates issued centrally – e.g. by the Request to Pay scheme, and that can be validated using associated PKI infrastructure (e.g. OCSP / CRL).

In the case of a participant providing end-user application, and where end-user identity is maintained by another participant (e.g. a message repository operator), the implementation shall be based on oAuth2, a web-standard based delegated authority mechanism. Specifically, the "authorisation code" grant type flow should be used, covering cases 3 & 4 in the list above.

5 Security Approach

5.1 Trust Management

The basis of trust is an offline accreditation process of Third Party Providers - participants operating message repositories and applications. Ideally, each TPP should be accredited using UK Open Banking scheme. On successful OB accreditation, the TPPs can then apply for Request to Pay accreditation. In this manner, most of the lifecycle events such as suspension or deregistration can be handled using existing processes in OB.

Trust revocation can be achieved in two ways. Revoking OB registration status is likely to involve revoking the certificate issued by OB. In addition, Request to Pay status revocation can be achieved by modifying TPP status in the Index. An API which allows TPP registration status check is provided for this purpose.

5.2 Securing Data in Transit

Data-in-transit is to be secured by standard TLS.

All message exchanges between third parties, and between third parties and central infrastructure shall be encrypted via TLS and authenticated via mutual TLS certificate authentication, relying on supporting PKI infrastructure as described in section 6.2.

All message exchanges between end-user applications and their supporting infrastructure shall be encrypted via TLS.

As an additional security measure, each hop in message delivery would embed a metadata entry containing TPP name and certificate fingerprint presented by the sending side.

5.3 Securing Data at Rest

The risk profile for Request to Pay data at rest is relatively low compared to traditional payment systems. However, due to stringent Data Protection regulations within UK and EU, third party providers are advised to encrypt data at rest.

Data at rest in the central infrastructure must be encrypted.

5.4 Message Authentication

Message authentication may be used to ensure message integrity between the sender and receiver. While this function is currently under review, there are several possible ways to implement this:

1. Sender embeds a one-way hash function of the message content which can used by the receiver to ensure massage has not been tampered with. Example hash functions are SHA-512 from the SHA2 family
2. A one way hash function (as above) is embedded into the message each time the message is transported between service endpoints (e.g. App-Repository, Repository-Repository).
3. Using HMAC message authentication algorithms where sender would generate a HMAC based on a shared key that the receiver can recreate.

While options 1 & 2 are similar and ensure messages were not tampered with in transit. Option 3 is qualitatively different in that it may also ensure senders identity, however would also require a way to securely transport keys between the Payee and Payer which may be very difficult for non-connected end users.

5.5 Common Attack Vectors and Mitigation

5.5.1 Biller impersonation

In this attack scenario, the attacker would attempt to impersonate a valid Biller, known to the Payer.

Biller impersonation is difficult considering the chain of trust in the present Request to Pay design, and perhaps only possible in the case where a TPP certificate is compromised (e.g. biller app or repository certificate is stolen).

There are several potential mitigations for this type of attack.

A standard feature of Request to Pay is the use of Confirmation of Payee (CoP) to ensure Payer can verify the payee's account. Further, an approach is presented whereby Payers will be able to generate per-biller aliases. This would help identify a situation where biller doesn't match the alias and raise warning.

Finally, use of token-based biller authorisation has been suggested. This is still under consideration as it may impact operation of service for cash payments or non-connected users.

5.5.2 TPP impersonation

In this scenario, the attacker may attempt to impersonate a valid repository or a valid biller application so send fraudulent Requests to Pay.

Present solution guards against TPP impersonation by having an offline accreditation system and online, real-time revocation facility. This approach is consistent with industry standards and PSD2 implementations.

5.5.3 Message tampering and MITM

In this scenario, attacker would intercept valid biller messages and modify content of the message to, for example, change target account.

Industry standard practice is to introduce message authentication codes (MAC) to authenticate messages, which will be applied in Request to Pay.

6 Functional Description

6.1 Overview

Request to Pay aims to provide UK consumers with a modern, advanced and extensible payments messaging service, embracing broader market and creating an ecosystem for applications that utilise the services.

Request to Pay can be can be implemented as a messaging service – akin to email - overlaid over existing payments infrastructure. As in most messaging services, we envision following core components:

- A single central index, enabling registration of various participants and discovery of services they provide
- A third-party operated message repositories, enabling end-users to store messages, as well as playing a role in sending and receiving messages.
- A third-party operated "client" applications, able to communicate with repositories to send and retrieve messages.

The overall, high-level solution is presented below.

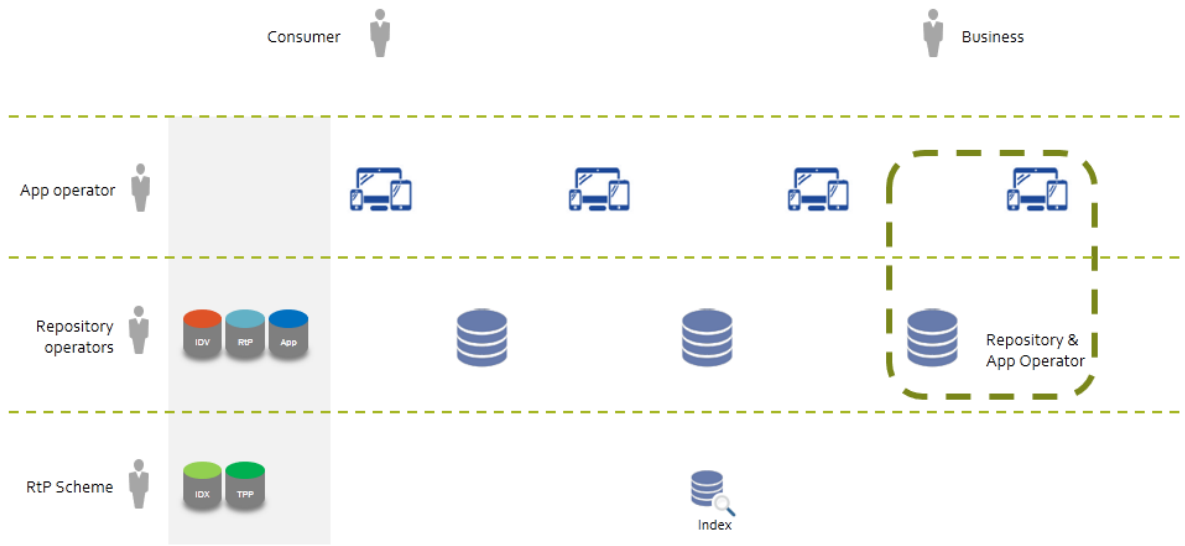


Figure 4 High-level solution architecture

It is worth noting that a general purpose, secure messaging service can have multitude of uses beyond Request to Pay. While every effort has been put into ensuring generality of the solution, further uses are not enumerated or analysed here.

6.2 Third-party Provider Identity and Onboarding

Third parties can participate in Request-To-Pay service in two capacities:

- As end-user application providers (systems of engagement)
- As repository service providers

The two types of third parties have a set of common attributes (including criteria for onboarding), and additionally, a set of attributes that apply to each category separately.

In the Request to Pay service, both types of third parties are identified using Public Key Certificates supported by central PKI infrastructure. The certificates are issued as a result of successful third-party onboarding process, and can be revoked at any time by the issuing authority. Issuing authority must operate highly available OCSP or CRL service in order to ensure identity validation.

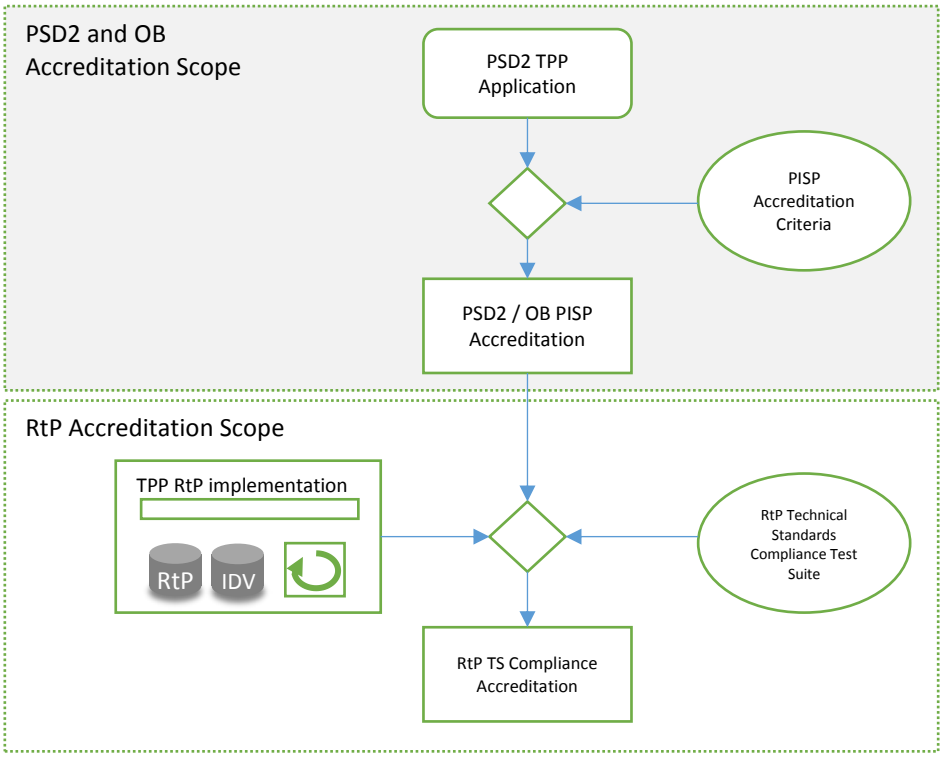


Figure 5 PSD2 and Technical Standards accreditation scope for Request to Pay providers

Onboarding and managing third parties requires a clear set of rules, as well as a governance process to ensure smooth operation of the service. These are to be defined as part of terms of service and contracting with third parties.

An example of the onboarding and governance, including infrastructure to support these processes, is the implementation of UK Open Banking and PSD2 PISP third-party management. This system can be used as a basis for Request to Pay third party management.

Mechanisms for establishing third party identity, credibility, regulatory compliance and application of security standards can be used directly by relying on PSD2/OB registration and underlying PKI infrastructure. Lifecycle events such as suspension or loss of accreditation, disputes and arbitration etc. would all remain in the domain of the PSD2/OB regulatory bodies, and outside of the Request to Pay service scope.

Beyond requirements described above, third parties implementing repositories are also required to pass a suite of technical requirements. This test suite would exercise the technical implementation of the APIs, security and resilience of their implementation.

6.3 End-user Identity

Request to Pay as a service depends on ability to uniquely and securely identify end users using the service. The "Primary Identifier" would ensure messages can be routed to the right end user – be it Payee or Payer.

In an analysis exercise, a number of Primary Identifiers have been considered, outlined below. Importantly, the service must cater for a variety of users that may not have internet connectivity, smart devices or bank accounts.

	Mobile number	Email Address	Username	Abstract number	Bank account # / SC	Proxy ID	Name & address
User Experience	●	●	◕	○	◔	●	◔
Support non-banked end users	●	●	●	●	○	●	●
Support non-connected end users (no mobile)	○	●	●	●	●	●	●
Support non-connected end users (no internet)	●	○	◔	◕	●	●	●
Support non-connected end users (neither)	○	○	◔	◕	●	●	●
Ease of integration with PayM and other schemes	●	◑	◑	◔	◕	●	◕

Table 1 Primary Identifier for various users

It is worth noting the different expectations on the identity of the Payee (e.g. biller) and Payer. While it is critical that the Payee identity can be validated, Payer identity verification may in some cases be of lesser concern.

Further, it is worth noting that there is an expectation that consumer may only be able to issue a Request to Pay if they have a bank account or equivalent way to receive electronic payments. There is no such expectation in the case when consumer receives and pays a Request to Pay.

From the consumer point of view, it is also very desirable to make it possible to use an existing primary identifiers consumer are used to – such as mobile number and email.

In order to satisfy a range of requirements while also ensuring broadest applicability to various types of users with varying access to banking services, internet and smartphones, a scheme based on proxy-ID has been adopted.

The Primary Identifier (PID) is composed of:

- The user ID (unique within a provider repository)
- The hash character #
- Repository ID

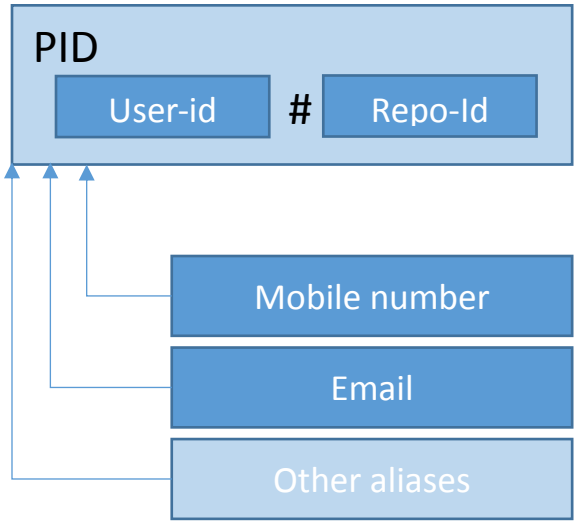


Figure 6 Primary Identifier

The user may also associate a number of aliases with their primary identifier. A mobile phone number and email address are two types of aliases that may improve end-user experience and usability. Other alias types can be defined.

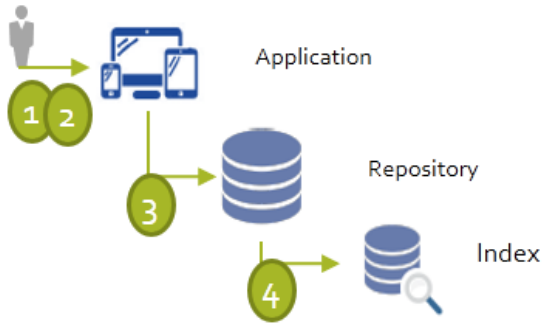
Aliases may also be used by Payer applications to maintain control over who can make Requests to Pay. For example, by generating an alias for each Payee, the Payer may have more control over who may issue requests.

6.4 End-user Onboarding

We envision three fundamental scenarios for end-user onboarding.

6.4.1 Onboarding based on existing relationship

In this scenario, the end user has an existing relationship with a repository operator. For example, a bank, Personal Finance Management provider, a utility provider (or more generally a business with which the consumer is registered), may be accredited as a repository operator. In this scenario, the end user may opt-in to use the Request to Pay service with this provider.



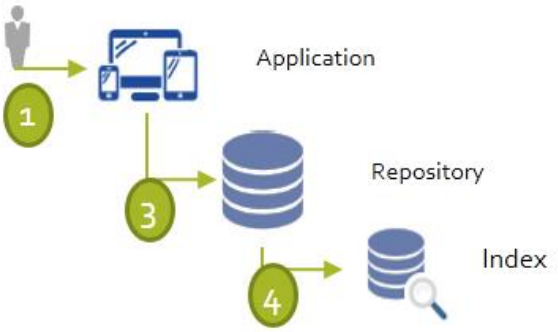
1. User is logged in using existing credentials
2. User agrees to add RtP service to their existing services.
3. A PID and message-box is created for this user by the Repository.
4. User can choose to associate their mobile number or email with this account

4.1 In this case, a record is stored in the index to map mobile to PID.

Figure 7 Onboarding based on existing relationship

6.4.2 Onboarding to a new relationship

In this scenario, the user learns about Request to Pay and independently obtains a means to access the service – e.g. downloading an App from an App Store, or creating a new registration with a Request to Pay provider.



1. User registers with TPP application
2. To receive payments, a bank account is required. Confirmation of Payee can be used along with Identity documents.
3. A PID and message-box is created for this user by the Repository.
1. User can choose to associate their mobile number or email with this account

4.1 In this case, a record is stored in the index to map mobile to PID.

Figure 8 Onboarding based on new relationship

6.4.3 Onboarding via invitation from Payee

In this scenario, an end user receives a bill where a Request to Pay is provided as payment option. Such bill would contain an embedded invitation link (e.g. QR or BAR code) that can be used to initiate registration process.

From this point on, this case is equivalent to "Onboarding to a new relationship" as described above.

This may mean that Payee operates the application and repository, it may be a third-party repository that Payee has selected, or the Payee may, in their registration process, provide a choice of TPP operators to the consumer.

6.5 State Management

A common query regarding Request to Pay is who owns the ultimate state of a request – e.g. current balance, settlement status etc. One specific question in this area is when would the system consider a request paid – when the Payer makes the payment, or when the Payee settlement is complete.

Conceptually, Request to Pay is a messaging service, not a payment service. As such, Request to Pay does not seek to modify the way either Payer or Payee manage their state. Indeed, Payer may consider the bill paid once the payment is made, while Payee will consider the bill paid once the payment is recognised on Payer's account. Request to Pay does not modify this.

The state of relevance for Request to Pay service is to ensure both Payer and Payee see exact same messages for each Request to Pay. Solution must ensure that each hop in message delivery provides receipts and clear responses in case of failure to deliver messages. Further, each participant in the exchange may at any point request any other participant adjacent in the delivery chain to synchronise messages for a specific Request to Pay.

Further a need was identified to understand when a request is "closed", under which conditions and what end states of the request are possible.

End states

General approach is that there are two way to close the request:

- Payment made in full
- Request declined
- Payment period ends

Following end-states are currently under consideration:

- Paid fully
    - Description: The request is paid by the Payer
    - Typical cases: The full amount is paid before the end of the payment period (either with a single Pay-all payment or multiple Pay-partial payments)
- Paid partially
    - Description: The request is partially paid, but the payment period has expired
    - Typical cases: While a partial payment has been made, payment period has ended without full payment.
- Not paid
    - Description: The payment period has expired without a payment being made
    - Typical cases: Payer ignored the request and has not made a payment.
- Rejected
    - Description: The Payer has rejected the request
    - Typical cases: Payer did not recognise the charge or has made the payment outside the RtP system.

It is worth noting that requests will only ever be soft-closed. That is to say, no data is deleted and no irreversible action is performed on the request; the "message thread" remains, and both biller and payer are able to post new messages (e.g. contact biller / contact payer would still be possible).

In effect, the biller and payer applications would determine the status of a request by analysing messages in the thread.

<u>Payer end state (without biller reconciliation)</u>

From the Payer's perspective, requests are "closed" based on Payer actions (e.g. pay, decline etc.). There are several good reasons for this.

A. Payer applications would only send a message when a payment (to the account specified by the Biller) is successful.

B. Request to Pay ensures that Payee account is part of the request, and is validated using ANVS, and therefore it is very unlikely that the payment would be made to wrong account.

If a biller determines that the payment has not been made (despite RtP message), biller should have a facility to reopen the request via additional Request to Pay message. This is preferred to contacting the payer out-of-band or activating delinquency process.

Biller end state (with biller reconciliation)

As discussed, biller end state is determined by biller's accounting systems.

If we take the position that biller acknowledgement is required to "close" the Request to Pay for biller – for example after account reconciliation on biller side - biller application would be required to detect payment of a particular request and send a message to acknowledge payment.

In this case, a paid request would remain open (or perhaps somewhere between "open" and "closed") until acknowledgement is received, even if payment period ends. This also has ramifications on the "Paid partially" end state – where the request would remain open until biller confirmation of the partial payment.

Main benefit of this approach is that biller state is accounted for in "closing" the request. However, this also has some drawbacks.

- Should this interim state be exposed to end users (payer in this case) - this may seem odd to the end-users who are not used to having visibility of biller settlement. Could be particularly difficult in case settlement takes days or weeks.
- This may increase complexity of biller applications as they must send messages to acknowledge payment for each payment on each request.
- This would increase the complexity of entering end-states – as reconciliation may take any amount of time, closing of requests may need to be delayed beyond end-date (also applies to partial payments where we may need to wait for multiple settlement messages).

6.6 Resilience to Failures

The approach taken is that failures are part of Business as Usual (BAU), rather than extraordinary events.

6.6.1 Out of sync repositories

In a messaging system as the one being described, it is possible to enter a state where sender and receiver message boxes for a specific Request to Pay do not match.

It is worth noting that this should not be possible during normal operation of the systems. Each party in the delivery chain receives an immediate API response code which signifies success or failure of completing the hop, and in case of failure the delivery failure is propagated back to the sender.

However, the overall service must maintain resilience in case of failure of any component. For example, if a biller repository suffers data loss after receiving a message, the two repositories may have conflicting information about the delivery of a said message. And, as discussed above, each participant in the exchange may at any point request any other participant adjacent in the delivery chain to synchronise messages for a specific Request to Pay.

6.6.2 Repository operator ceases operation

Repositories may become temporarily unavailable through either suspension of their accreditation, technical failure or for other reasons.

A set of rules is necessary to govern behaviours in this case in order to ensure minimal disruption and recovery mechanisms for end users. As the very high level principles, the following should be considered:

- All message delivery attempts (either from app or other repositories) must return to sender.

    - This ensures the other party can find out when first party is unable to receive messages.
- Applications must display end-user notification that their message box is currently unreachable by billers
    - This ensures the first party is aware they are unable to send or receive messages.

In the event of accreditation suspension or when the repository permanently ceases operation, all alias values should be purged from the Index, allowing end-users to re-register their mobile numbers and email addresses to PIDs hosted by another repository.

6.6.3 App operator ceases operation

An app operator may cease operation through either suspension of their accreditation, temporary technical failures or permanent closure of business. In this case, end-users can easily switch to another App provider without losing their PID or any messages – as the repository maintains their message box.

6.6.4 Index failures

Failure of index functions may be critical for the operation of the system. While every effort is required to ensure high-availability of the index – with multi-cloud, active-active redundancy implementation, a closer look is required at each service provided by the index, its impact on the overall system operation and mitigation techniques.

6.6.5 Rejection failures

It is worth considering that end-user account lifecycle must include account decommissioning – e.g. when the user de-registers from a repository. In this case, end-users PID stops being valid on present repository, however other participants may still attempt to send messages addressed to decommissioned PID.

In this case, the recipient repository would reject message by returning a specific error and the message must be returned to sender.

6.7 Executing a Payment

Payment execution is within the operational scope of end-user applications. The end user application may present the user with a range of payment methods, and execute the payment using any electronic payment system appropriate.

It is worth noting that the method of payment does not have substantial architectural impact on the operation of Request to Pay service. Execution of payment is decoupled from messaging. However, key requirement is that messaging formats in Request to Pay facilitate transfer of information necessary to initiate and execute payments.

Distinct cases to consider are outlined in more detail below.

6.7.1 Non-PSD2 Electronic Payment

A typical scenario for non-PSD2 electronic payment is when a bank operates as a Request to Pay provider. In this case, bank's standard payment user journey and underlying infrastructure (e.g. faster payments) may be used to execute a payment without re-authenticating the user.

User Journey

The mock-up of the user journey is presented below.

Figure 9 Request to Pay user journey

6.7.2 PSD2 electronic Payment

In this scenario, the Request to Pay Payer application is also a PSD2 PISP.

Payment Initiation Service Provider (PISP) is a new category of a service operator created by the EU Revised Payment Services Directive (PSD2). The regulation allows a PISP to register multiple payment accounts for an end-user and initiate payments using these accounts, defining rules to govern these interactions. For each transaction, the PISP is required to redirect user to the bank, and subject user's consent given to the bank, the bank would issue a token to the PISP allowing them to initiate the transaction (so called dynamic linking). In essence, the payment user journey would involve these three parties at a minimum (end-user, PISP and the bank).

User Journey

The mock-up of the user journey is presented below. In the mock-up, the "Payment Pal" is the Request to Pay app operator that is also a PISP, "GB Gas" is a biller, while "UK Bank" is a bank that user selects to pay from.

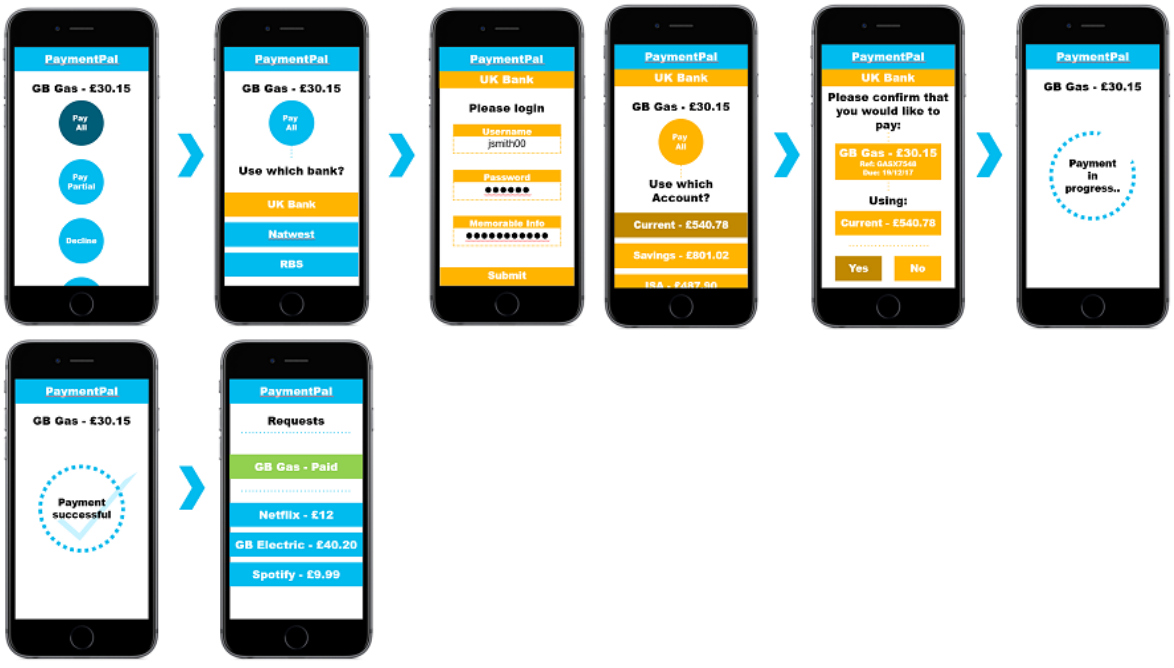


Figure 10 Request to Pay PaymentPal user journey

Sequence flow

The execution flow for PSD2 payment is presented below. PSD2 roles for swim-lanes are noted in Amber. For simplicity, multiple bank endpoints (/authorise, /token, /payment) have been collapsed into a single swim-lane, without losing generality of the sequence flow.

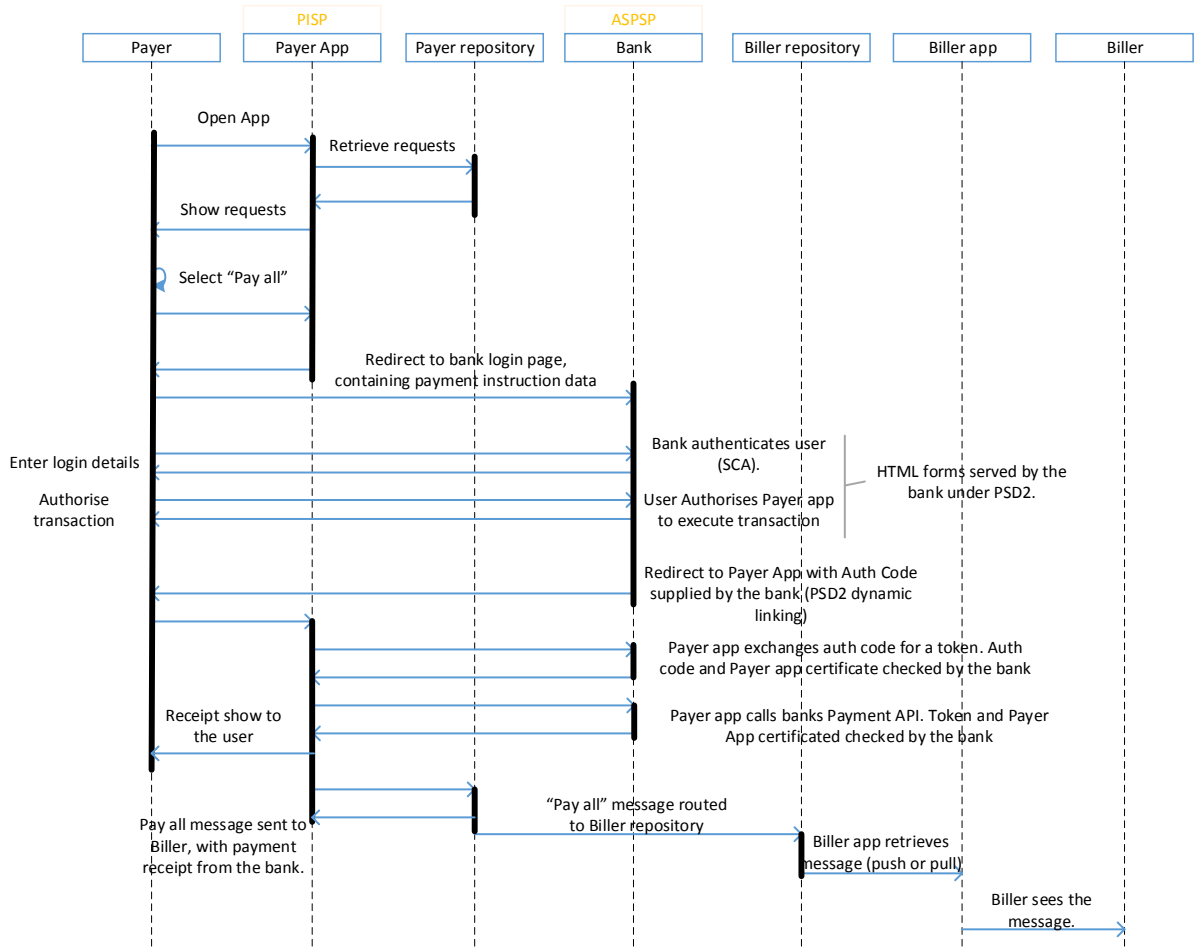


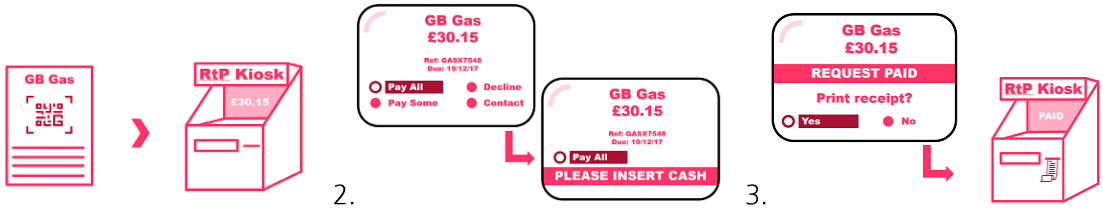
Figure 11 Execution flow for PSD2 payment

6.7.3 Cash payment

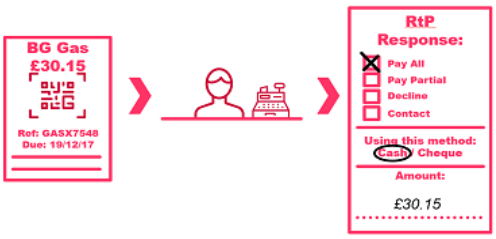
User Journey

Cash payment is a crucial facility required by the service. The approach is to enable end-users to interact with the Request to Pay service using a counter or self-service kiosk facilities. In either case, the counter operator or the kiosk machine are taking the role of Payer Application in the Request to Pay system, and are sending messages / responses as the Payer Application would.

The user journey for self-service kiosk:



The user journey for counter service:



6.8 Request to Pay walkthrough

In this section, we will walk through processing of a single request to pay. For simplicity, the end-user-onboarding scenario presented here is simplified. A more complete overview of this topic is discussed in detail in section 6.4.

Stages in processing are presented in Figure 4.

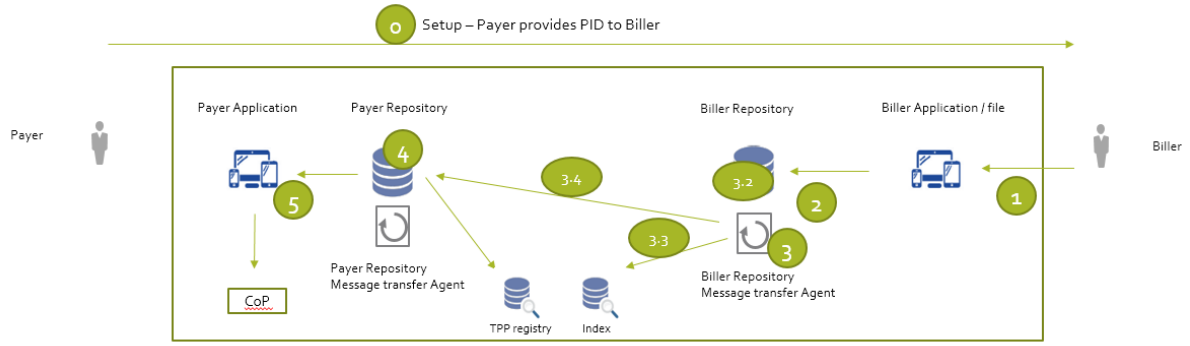


Figure 12 Solution walkthrough

Setup (step 0) - As part of contracting for the service, Payer provides their Primary Identifier (PID) to the biller.

1. Biller generates a RtP and submits it using "Biller Application"
    1. This may be either bulk submission, individual submission or "file"
2. "Biller Repository" queues the requests (messages)
3. For each request in the queue
    1. Biller repository looks at Payer PID, extracts Repository RID
    2. Messages addressed to local (on-this-repo) users are delivered to local message-boxes
    3. Based on Repo_ID, look up API endpoint (target repository) in the Index
    4. Messages addressed to remote (out-of-this-repo) users are sent to target repository
4. "Payer Repository" verifies sender, repository and stores message in the payer message-box
5. "Payer Application" retrieves messages from "Payer Repository" – either push or pull

6.9 Functional requirement summary

In this section, we enumerate functional requirements established to date. This section is work in progress.

Category	Number	Functional requirement
General	G1	Request to Pay implementations must be invariant to payment methods.
TPP	T2	Allow Third-Party operators to operate end-user applications
TPP	T3	Allow Third-Party operators to operate message repositories
TPP	T4	Provide a central Third-Party registry that provides standard PKI infrastructure, including issuing certificates and real-time certificate revocation checks.
End-user	E1	Allow end-users to register with Third-Party providers
Repository	R1	Support transporting messages between Payer and Payee who may have different service providers (third party operators).
Repository	R2	Support any number of messages associated with a specific request to pay, and easy retrieval of all message associate with a specific Request to Pay.
General	G2	Support message attachments so that a bill may be attached to a request, or a receipt may be attached to responses.
General	G3	Support validation of message format and type in the context of original request, to provide assurance message can be understood by the other party
Index	I1	Support validation of Payee details by Payer – e.g. via assurance data
General	G4	Support transporting messages between Payer and Payee who may have different service providers.
Index	I2	Support mutual authentication between any two third-party providers for each transaction between them.
Index	I3	Support mutual authentication between central infrastructure and third party providers for each transaction.
Index	I4	Support real-time checks of Third-Party provider identity
General	G5	Support message authentication codes that will prevent tampering with a message in the delivery path.
General	G6	Each participant in message delivery (repository, application) must provide confirmation of message receipt and notification of failure to deliver a message.
General	G7	Each participant in message delivery (repository, application) must provide a facility to retrieve all messages associate with a specific Request to Pay
General	G8	Support uniquely identifying end-users based on a PID
General	G9	Support email and mobile phone number as PID aliases
General	G10	Support end users who do not have a bank account as Payers
Application	A1	Third-Party Application providers must support at least one electronic payment method
General	G11	The service implementation must not exclude cash payments, however Third Party provider may choose not to provide this facility
General	G12	The service implementation must not exclude end-users who do not have smart devices
General	G13	The service implementation must not exclude end-users who do not have internet connectivity
Application	A2	Request to Pay may only be generated with a full PID of the Payee
Application	A3	Request to Pay may only be generated if a Payee has a bank account on file
General	G14	Associating a Payee with a bank account must involve Confirmation of Payee check
Application	A4	Payer must clearly see the result of Confirmation of Payee query on the Payer
Index	I5	When looking up an alias, Index must not disclose PID but must rather return only the Repository endpoint for the given alias.

Table 2 Functional requirement summary

7 Architecture Blueprint

7.1 Overview

Overall service architecture is presented in the diagram below.

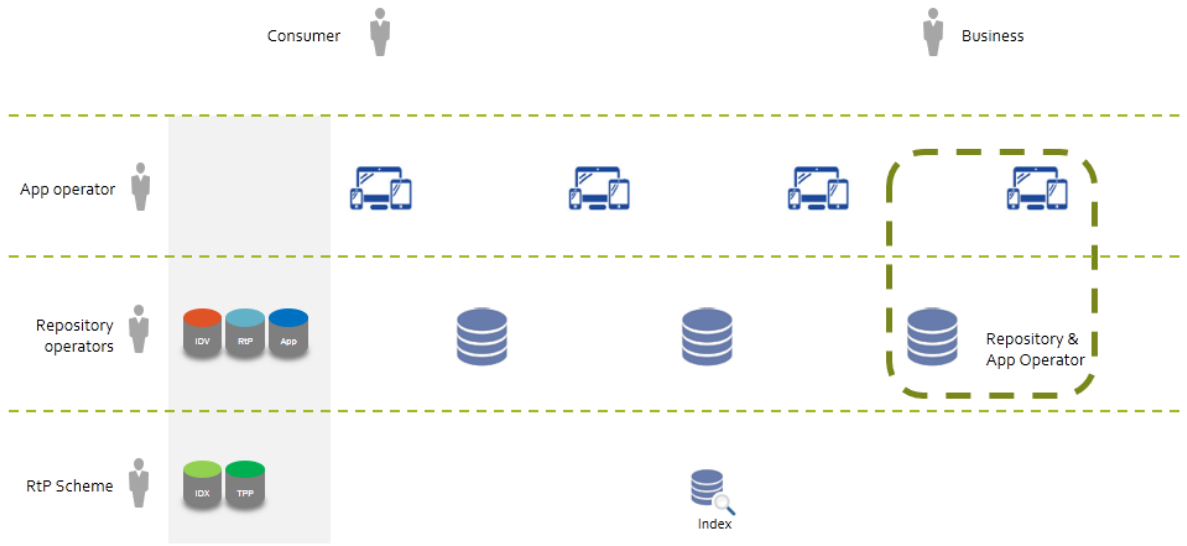


Figure 13 Service Architecture

The presented solution is a messaging system, overlaying payments, while being agnostic of payment method. End users use client applications to send and receive messages. Messages are passed to and stored in Message Repositories, which also take part in transporting messages between repositories that hold messages for different end users. The Index is used to discover API endpoints for repositories, and to maintain aliased Primary identifier. In many ways, this system is similar to email delivery system. Client applications are akin to email readers, and may allow end users to read messages from multiple message-boxes. End user accounts are held by repositories, in a similar manner to email where user identity is with the server. Central "Index" performs discovery functions, similar to the way DNS services are used to query MX records and resolve mail server addresses.

Key differences between this system and email are:

- Security is stepped up across the board and is intrinsic to the service design. Parties operating applications and repositories are accredited and all communication is over HTTPS by default. Other features (e.g. message authentication etc.) are also considered.
- Use of modern protocols (RESTful APIs over HTTP) as opposed to traditional SMTP and IMAP. This allows implementers to reuse broad range of modern off-the-shelf solutions, tools, architectural and security patterns for a variety implementation and deployment scenarios.
- Messages transported by this system are structured – as opposed to freeform payload of email messages. Messages in this system are machine readable, which enables automation of message processing (e.g. bot responders).

There are three essential components in the implementation of the system – Index, Message Repositories and End-user Applications.

Index provides discovery and accreditation functions, holding data about third party accreditation, internet addresses of repositories and end-user alias data.

Repositories hold end-user message-boxes, end-user Identity data and provide functions for both – inter-repository message delivery and APIs that support Applications.

Applications present user interface to data held in **repository**.

7.2 Message Structure

Messages are structured JSON objects, allowing machine processing and automation. Each message consists of several sections encapsulating:

- Message metadata
- Transport metadata
- Thread information
- Message content

Sample message with comments on types, typical values and value options for Request to Pay is shown below.

{

"threadHeader": { // To be added by the message creator (App) "profile": "RequestToPay",

"threadID": "user1-repo1-user2-repo2-timestamp_nano-64_bitsrandom", "requestType" : "new/existing",

"serviceDetails" : "string like Window Cleaning", "created": "<Time in millis>",

"mac": "47863587653C6674A24234A85878FFB727646"

},

"threadMeta" : { //To be added by Repository "senderPID": "dileep017#Repository1", "recipientPID": "seandoh#Repository1", "num-messages": 33

}

"messageHeader": { // To be added by the message creator (App) "messageId": "timestamp_nano-64_bitsrandom", "created": "<Time in millis>",

"senderPID": "dileep017#Repository1", "recipientMobileNumber": "07825853192", "recipientEmail": "string",

"mac": "47863587653C6674A24234A85878FFB727646"

},

"messageMeta" : { //To be added by App or Repository with whom this info is available "senderName": "dileep g",

"recipientPID": "seandoh#Repository1",

}

"deliveryMeta": { //To be added by Repository

"deliveryStatus": "<arrived/delivery-succeeded/failed-delivery-($failed-reason)>", "deliveryPath": [{

"from": "192.0.0.1",

"to" : "<RepositoryID>" "repositoryType" : "<Sender/Receiver>" "senderCert": "C6674A24234A85878FFB",

"timestamp": 1404869611938,

}, {

}]

},

"from": "192.0.0.2",

"to" : "<RepositoryID>" "repositoryType" : "<Sender/Receiver>" "senderCert": "C6674A24234A85878FFB",

"timestamp": 1404869611938,

"messageBody": { // Generated by the message creator (App) example below for RequestToPay profile "messageType" : "PayAll| PayPartial| ReqPayExtension| Decline| DeclineBlock| NoteToBiller|

RequestToPay| ExtensionGranted| ExtensionDeclined| NoteToPayer", "paymentAmoutRequest" : "number",

"currency" : "GBP",

"paymentDueDate" : "Epoch time in millis", "extensionDateRequested" : "Epoch time in millis", "paymentRequestedDate": "Epoch time in millis", "accountDetails": {

"accountNumber": "Sender's Account Number", "sortCode": "Sort Code of the Bank"

},

"paymentAmout" : "number", "paymentDate" : "Epoch time in millis", "note" : "Note by Sender", "Attachments": {

[{

"file-name": "file1.pdf",

"content": "base64",

"mac": "2837648275878FFB727646"

}]

},

"mac": "28376482736C6674A24234A85878FFB727646"

}

}

7.3 Message Flows & Sequence Diagrams

7.3.1 Registering a new user

A registration scenario corresponding to Section 6.4.2 (using App store to install RtP app) is presented in the sequence diagram form below.

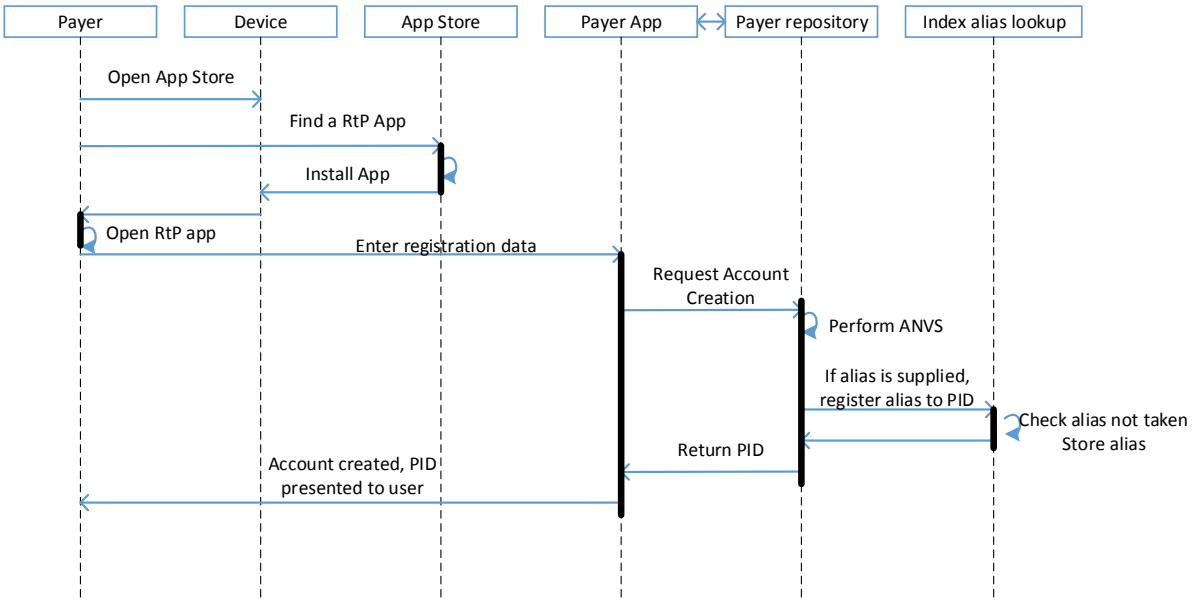


Figure 14 Sequence diagram to show Request to Pay installation using App store

7.3.2 Message exchanges

Messages sent by the biller, and especially initial request delivery, require a high standard of trust in the message delivery chain. For this reason we will demonstrate delivering initial Request to Pay below, but without losing generality as this pattern is reusable for delivering all biller messages.

Delivering a request to pay involves a range of architectural components and interactions between them as shown in the diagram below.

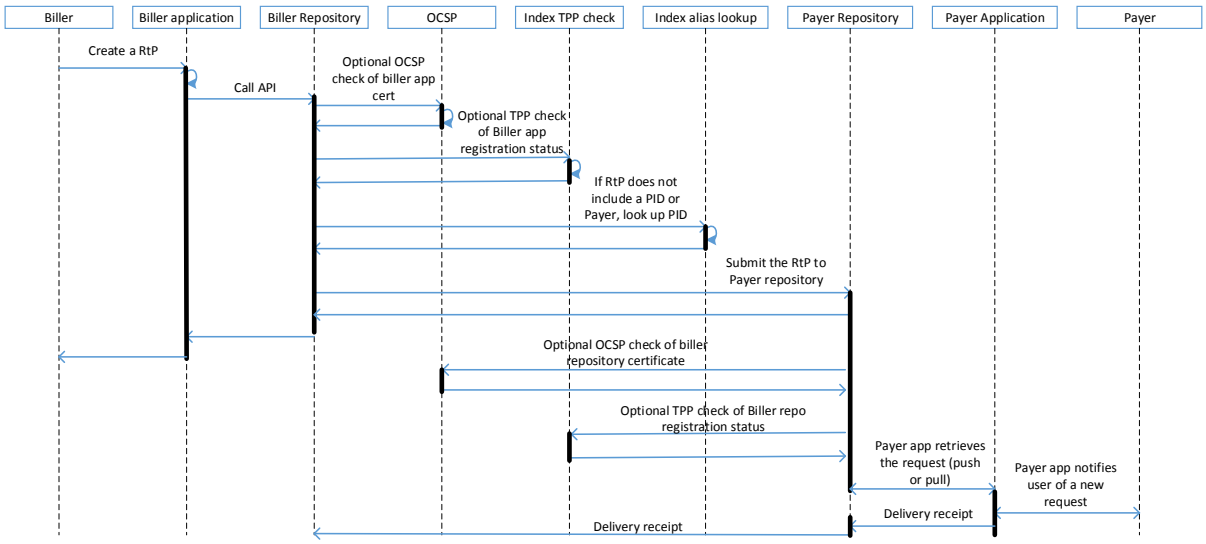


Figure 15 Request to Pay message exchanges

Note that optional OCSP and TPP checks, along with alias lookup are not expected to occur for every request. Values can be cached to prevent excessive per-request traffic, with short cache TTL ensuring small window of opportunity for revoked / invalid TPPs to interact with the service.

Delivery receipts are optional. If a receipt is requested by any sending party (app, repository), all parties downstream in the delivery chain must send delivery receipts.

The execution flow for Payer messages closely resembles the sequence above.

7.3.3 Delivery failures and state synchronisation

As discussed in section 6.6, failures of any component in the system is treated as Business as Usual (BAU), rather than extraordinary events. Following cases are to be explored and documented:

- Validation failure
  - TPP registration check offline
  - TPP registration check negative response
  - Invalid / revoked certificate presented
- Alias resolution failure
  - Index alias service offline
  - Alias not found
- Sender repository failure
  - Sender repository offline
  - Sender repository rejects request
  - Sender repository presents invalid certificate
- Receiving repository failure
  - Receiving repository offline
  - Receiving repository rejects request
  - Receiving repository presents invalid certificate

General approach is as follows:

Failure type	Handling
Registration check Certificate failures	Message must be returned to sender. Sender must be presented with a failure report and remediation instructions depending on the cases as described in section 6.6.
Service-offline failures	Sending party agent (app or repository) must retry sending the message. A retry cycle pattern may be 5min, 15min, 1hour, 6 hours, 24hours, abandon. Sender must be made aware that message delivery is pending.
Rejection failures	Rejected messages should return to sender, as described in section 6.6.5.

Table 3 Failure types and handling

7.4 Components

In this section, we will propose implementation scenarios for main components of the Request to Pay service – Index, Message Repository and Applications.

7.4.1 Index

Overview

The high level architectural diagram for the Index component is presented below.

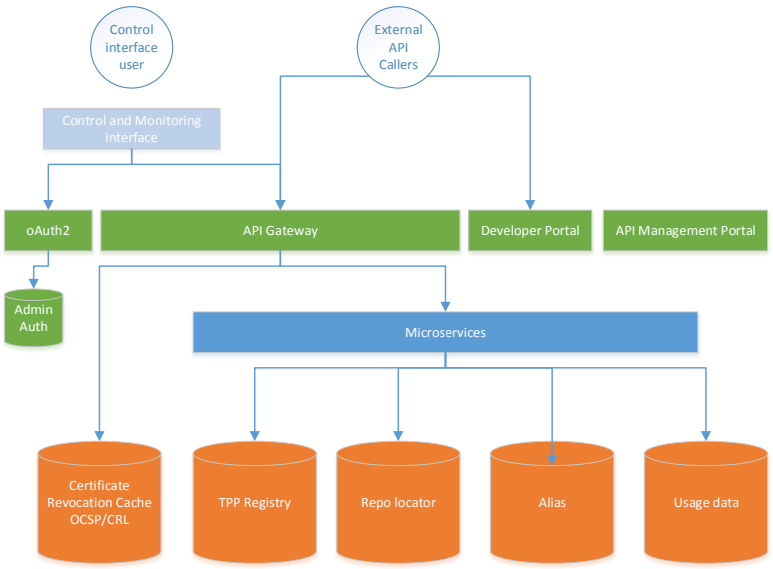


Figure 16 High level architecture for the Index

Index is essentially a data lookup service that enables validation of Third Party Providers registration status, discovery of TPP API endpoints and lookup of end-user aliases. As a centrally operated component, the index may collect Quality of Service (QoS) data, provide system control and analytics functions.

Index is implemented in a layered architecture. Data layer (amber in the diagram) is exposed via Microservices (blue layer). API Platform (green components) exposes the services to the scheme participants, and provides several standard API platform services. The services are consumed by TPP operators, while the command and control web application (light blue) is available data **management**, system control and analytics.

Functional description

Discovery services

The Index provides following discovery services to Message Repositories:

- TPP Registration check (given a third-party ID and certification fingerprint, respond with registration status)
- Alias lookup (retrieve PID based on alias such as mobile number)
- API endpoint lookup given a RID (retrieve API URL given repository ID)

Data services

Further, to ensure operation, following CRUD (Create, Retrieve, Update, Delete) services are required:

- Management (CRUD) of TPP Registration records
- Management (CRUD) of PID aliases

Certificate revocation cache

This component ensures the API gateway can quickly check internal revocation cache, rather than executing a remote check against e.g. central open banking OCSP / CRL.

Control and Service Monitoring

As part of central infrastructure, the Index must also provide a management interface to control various aspects of the service. The management interface will require service monitoring and analytics and therefore, the Index must also provide functionality to capture usage and Quality of Service (QOS) data:

- Usage data API endpoint
- QoS report API endpoint

External interface APIs

All API calls are authorised by validating the certificate presented in TLS mutual-authentication.

The basic input and output data for the APIs is defined below.

TPP Registration check

API		Endpoint	
TPP Registration Check		GET /tppregistrationchk/<id>	
Inputs		Outputs	
URI Path: id	TPP ID to check	200 OK JSON Data	TPP data provided, including certificate fingerprint, App oAuth redirect URL etc..
Headers		Errors	
caller-id	The unique ID of the caller Repository of this API	404 Not found No data 400 Bad request 401 Unauthorized 500 Internal Server Error	TPP invalid Malformed data supplied Caller Certificate Invalid Server Error
type	Type of TPP to be checked (App / Repo)		

Alias lookup

API		Endpoint	
Alias Lookup		GET /alias/<alias>	
Inputs		Outputs	
URI Path: alias	The alias to get the PID for	200 OK JSON Data	Alias has been found and the PID is returned.
Headers		Errors	
caller-id	The unique ID of the caller (App / Repo) generated at the time of registering the caller	404 Not found No data 400 Bad request 401 Unauthorized	Alias has not been found Malformed data supplied Caller Certificate Invalid

TPP API endpoint lookup

API		Endpoint	
TPP API endpoint lookup		GET /tppappendpoint/<repoId>	
Inputs		Outputs	
URI path: repoId	The unique ID of the repository whose App end-point is enquired	200 OK <endpoint uri>	Repo ID has been found and endpoint URIs is returned.
Headers		Errors	
app-id	The unique ID of the caller App of this API	404 Not found No data 400 Bad request 401 Unauthorized	TPP ID has not been found Malformed data supplied Caller Certificate Invalid

TPP Repo endpoint lookup

API		Endpoint	
TPP Repository endpoint lookup		GET /tpprepoendpoint/<repoId>	
Inputs		Outputs	
URI path: repoId	The unique ID of the repository whose Repo end-point is enquired	200 OK <endpoint uri>	Repo ID has been found and endpoint URIs is returned.
Headers		Errors	
caller-repo-id	The unique ID of the caller Repository of this API	404 Not found No data 400 Bad request 401 Unauthorized	TPP ID has not been found Malformed data supplied Caller Certificate Invalid

TPP Repo Registration Check

API	Endpoint
check the Registration of a Repository	GET /tppreporegistrationchk/<repoId>

Inputs		Outputs	
URI path: repoId	The unique ID of the repository whose registration needs to checked	200 OK	Empty Response
Headers		Errors	
caller-repo-id	The unique ID of the caller Repository of this API	404 Not found No data 400 Bad request 401 Unauthorized	TPP ID has not been found Malformed data supplied Caller Certificate Invalid

Manage aliases

API		Endpoint	
Manage aliases		POST, PUT, DELETE /alias/<alias>	
Inputs		Outputs	
URI path: alias	Mobile number or email	200 OK <PID>, alias	Alias has been created, modified or deleted
PID	The PID to associate with alias		
Headers		Errors	
		500 Server error	Alias modification error

Usage data and Quality of Service

API		Endpoint	
Consolidated QoS Data		GET /qos	
Inputs		Outputs	
		200 OK JSON Data	Consolidated QoS data provided, grouped by Repositories, further sub grouped by QoS Type – USER, THREADS, MESSAGES, TRANSACTIONS
Headers		Errors	
		500 Server error	Error in getting data
		404 Not Found	No data found

Internal / management interface APIs

The APIs for internal / management console use are presented below. Use of these APIs are only allowed to clients presenting specific management console certificate.

TPP Registration Records

API		Endpoint	
Manage TPP Registration records of a TPP Get All TPPs, Create a new TPP		GET, PUT, DELETE /tpp/<tpp_id> GET, POST /tpp	
	Security:	Requires management console user authentication token	
Inputs		Outputs	
URI path: tpp_id	TPP id to retrieve, update or delete	200 OK JSON Data	TPP has been retrieved & returned, modified or deleted
JSON data: see TPP data schema	TPP data to Create	200 OK JSON Data 200 OK TPP data array	TPP Created All TPPs retrieved and returned
Headers		Errors	
		500 Server error	Internal Server Error
		400 Bad request	Malformed data supplied
		401 Not Authorised	If the certificate presented does not entitle the caller to execute this API

TPP App Registration Records

API		Endpoint	
Manage Registration records of a TPP APP Create a new APP		PUT, DELETE /app/<appId> POST /app	
Security:		Requires management console user authentication token	
Inputs		Outputs	
URI path: appId	App id to update or delete	200 OK JSON Data	TPP App has been modified or deleted
JSON data: see TPP App data schema	App data to Create	200 OK JSON Data	TPP App Created
Headers		Errors	
		500 Server error	Internal Server Error
		400 Bad request	Malformed data supplied
		401 Not Authorised	If the certificate presented does not entitle the caller to execute this API

TPP Repository Registration Records

API		Endpoint	
Manage Registration records of a TPP Repository Get All Repositories, Create a new APP		PUT, DELETE /repositories/<repoId> GET, POST /repositories	
Security:		Requires management console user authentication token	
Inputs		Outputs	
URI path: repoId	Repo id to update or delete	200 OK JSON Data	TPP Repo has been modified or deleted
JSON data: see TPP Repo data schema	Repo data to Create	200 OK JSON Data 200 OK TPP Repo data array	TPP Repo Created Repos retrieved and returned
Headers		Errors	
		500 Server error	Internal Server Error
		400 Bad request	Malformed data supplied
		404 Not Found	No Data Found (delete/update)
		401 Not Authorised	If the certificate presented does not entitle the caller to execute this API

Data model

This section provides **preliminary** look at possible data model for the index.

TPP data

Data required for Third-Party operators should include:

TPP data

Table – TPP data		
Name	**Type**	**Description**
TPP_ID	64bit int	Unique identifier of a third-party provider
Organisation name	String	Registered company name
Organisation reg number	String	Registered company number
Organisation address	Text	Address
Organisation telephone number	String	Telephone number
Organisation technical contact	String	Email address
Organisation administrative contact	String	Email address
Certification date	Date	Date of certification
Registration status	String	To be defined – provisionally one of: test, active, suspended, deleted
Repository operator	Boolean	If true, this TPP is authorised to operate repository

Application operator	Boolean	If true, this TPP is authorised to operate applications
Certificate fingerprint	Text	

Repository data

Table – Repository data		
Name	**Type**	**Description**
TPP_ID	64bit int	Unique identifier of a third-party provider
Repository ID	String	Registered repository name
Primary Repo API endpoint	String	Root URL for the API exposing repository-to-repository APIs
Secondary Repo API endpoint	String	Root URL for the API exposing repository-to-repository APIs
Tertiary Repo API endpoint	String	Root URL for the API exposing repository-to-repository APIs
Primary App API endpoint	String	Root URL for the API exposing application APIs
Secondary App API endpoint	String	Root URL for the API exposing application APIs
Tertiary App API endpoint	String	Root URL for the API exposing application APIs

Application data

Table – App data		
Name	**Type**	**Description**
TPP_ID	64bit int	Unique identifier of a third-party provider
App ID	String	Registered application name
App name	String	Registered application name
Primary oAuth redirect URL	String	Registered redirection URL for oAuth. See “End-user Authentication service” section in chapter 0.
Secondary oAuth redirect URL	String	Registered redirection URL for oAuth. See “End-user Authentication service” section in chapter 0.
Tertiary oAuth redirect URL	String	Registered redirection URL for oAuth. See “End-user Authentication service” section in chapter 0.

Alias data

Table – Alias data		
Name	**Type**	**Description**
Alias	String	Alias – e.g. Email or telephone number
PID	String	End-user’s primary identifier

7.4.2 Message Repository

Overview

The Message repository high level architecture diagram is presented below.

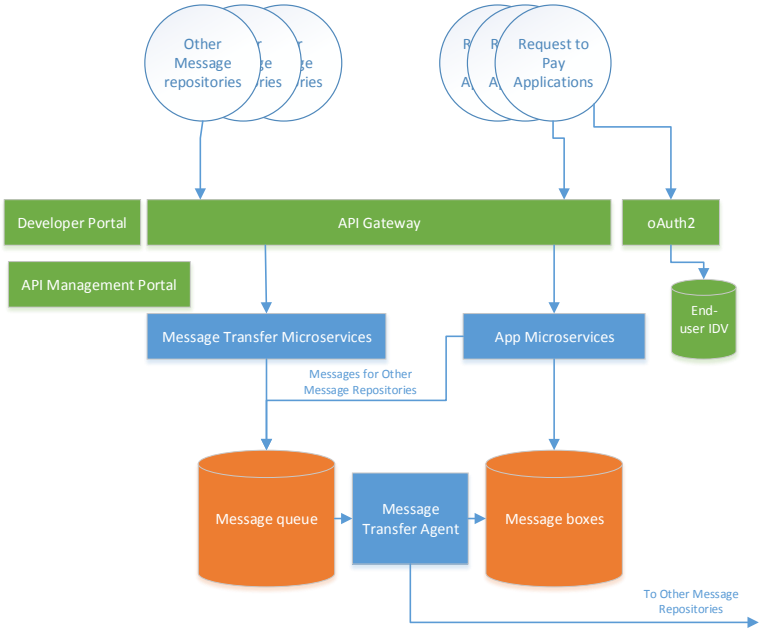


Figure 17 Message repository high level architecture diagram

The above is one possible representation of a repository implementation – there may be various implementations, so long as the External interface functions are compliant to Request to Pay specifications (to be defined). In this instance, a layered approach similar to the Index is presented, with the exception of the Message Transfer Agent component – a queue processing process which operates between the message queue table and message-box table.

Functional description

The message repository performs following functions:

- Provides end-user registration and authentication functions
- Maintains end-user message-boxes and provides authorised access to the messages
- Receives messages from other repositories (for locally serviced users only)
- Allows routing messages to other repositories
- Sends and receives message delivery receipts

In this particular design, the end-user registration function is provided as an API, so that registration can be implemented on a Request To Pay application.

Repository implementation is subject to a broad range of functional requirements described in chapters 4, 5 and 6. We will not re-state these requirements here.

All received messages are placed in a processing queue in the repository. A process (denoted as "Message Transfer Agent" in Figure 17) asynchronously retrieves messages from the queue and performs necessary actions. Actions may include:

1. For messages where receiver address is on the present repository, deliver message to local message-box
2. For messages where receiver address is on a remote repository, attempt delivering the message to a remote repository. If delivery fails, handle errors as described by resilience requirements in section 6.6. Should a retry be needed, return the message to the queue (with information on when to retry).

External interface – APIs

This section details APIs that repositories are required to implement in order to ensure interoperability with other service operators.

Application support APIs

All application support APIs are invoked over TLS and require mutual TLS authentication for validating the calling party. The message repository may at any point check validity of the presented client certificate.

End-user authentication service

This service allows end-users to use any app to connect to any repository. The App is essentially delegating authentication to repository, which in turn issues a token to the app. This token can then be used by the app to call APIs.

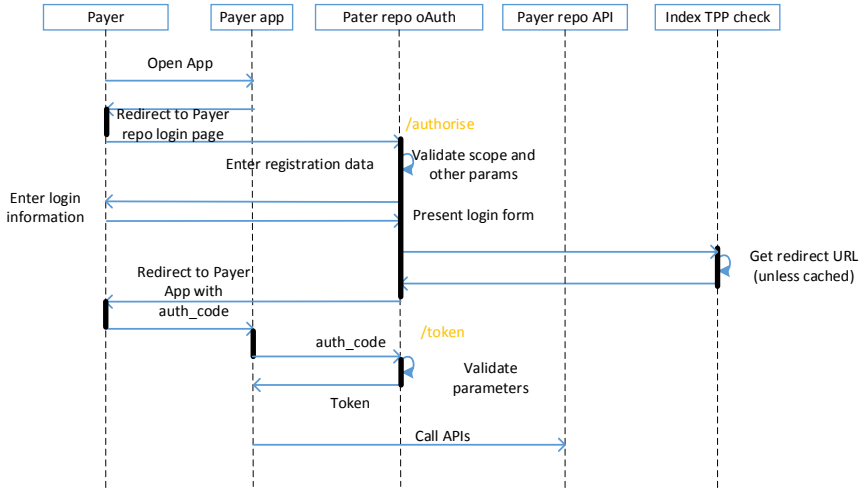


Figure 18 Auth2 applied for authenticating user from any app

End user authentication service must comply with RFC 6749 oAuth 2 specification and implement "Authorisation Code Grant" flow. The authentication initiation request /authorise, must be sent with following values of mandatory parameters:

Parameter	Status	Value
response_type	Mandatory	Value MUST be set to "code"
client_id	Mandatory	Fingerprint of the TPP certificate provided during accreditation
scope	Mandatory	Value MUST be set to "request-to-pay-app"
state	Recommended	Opaque value set by the client
redirect_url	Optional	This parameter can be used to select one of pre-registered redirect URLs listed in the index. If not supplied, the primary URL registered in the index will be used.

Table 4 Mandatory parameters for authentication initiation request

Note that any **redirect_url** supplied in the request can only be used to select one of pre-registered URLs, not to specify a new one. Values of **redirect_url** are centrally managed for security reasons.

List requests

API		Endpoint	
List my requests		GET /user/requests[?filter="sent"|"received"[&from=<fromval>[&to=<toval>]]]	
	Security:	Requires end-user authentication token (access-token)	
Inputs		Outputs	
filter	If parameter value "sent" is specified, list only Requests sent by the current user. If parameter value "received" is specified, list only Requests received by the current user. If parameter is not specified, list all Requests for the current user Any other value, return an error.	200 OK JSON DATA	JSON Array or Requests, categorized as sent and received RTPs and details of each requests.
from	UTC timestamp		
to	UTC timestamp		
app-id	The unique application Id		

profile	A string identifies it as a RTP request – always 'RTP'		
Headers		Errors	
		401 Not Authorised	If the certificate presented does not entitle the caller to execute this API
		400 Bad request	Invalid parameter supplied
		500 Server Error	Internal Server Error
		404 No data	No Data Found

Get messages for a request

API		Endpoint	
Get all messages for a request		GET /user/<RQID>/messages	
Security:		Requires end-user authentication token	
Inputs		Outputs	
Uri path: RQID	Unique ID of the Request to Pay	200 OK JSON DATA	JSON data: An array of Message objects containing messages associated with RID
Headers		Errors	
filter	If parameter value "sent" is specified, list only messages sent by the current user. If parameter value "received" is specified, list only messages received by the current user. If parameter is not specified, list all Requests for the current user.	500 Server error	Internal Server Error
app-id	The unique application id	400 Bad request	Malformed data supplied
from	UTC Timestamp	401 Not Authorised	If the certificate presented does not entitle the caller to execute this API
to	UTC Timestamp	404 No Data Found	No Data Found

Send a message

API		Endpoint	
Send a message		POST /user/<RQID>/messages?PID=[PID]	
Security:		Requires end-user authentication token	
Inputs		Outputs	
JSON data: A message Object to send	Request to Pay data	200 OK JSON Data	Message accepted for delivery. JSON data: RQID, Message ID
Uri query: PID	Primary user identifier sending the message		
Uri path: RQID	Unique ID of the Request to Pay		
Headers		Errors	
app-id	The unique application id	500 Server error	Send message error
		400 Bad request	Malformed data supplied
		401 Not Authorised	If the certificate presented does not entitle the caller to execute this API

Send a message (bulk)

API		Endpoint	
Send a message (bulk)		POST /user/messages?PID=[PID]	
Security:		Requires end-user authentication token	
Inputs		Outputs	
JSON data: An array of message Object to send	Request to Pay data	200 OK JSON Data	Message accepted for delivery. JSON data: An array of {RQID, Message ID}
Headers		Errors	

app-id	The unique application id	500 Server error	Send bulk messages error
Uri query: PID	Primary identifier of the user	400 Bad request	Malformed data supplied
		401 Not Authorised	If the certificate presented does not entitle the caller to execute this API

Register a new User

API		Endpoint	
Register a new User in Repository		POST /user	
Inputs		Outputs	
JSON data: User Object to register	New User Data	201 OK	
Headers		Errors	
app-id	The unique application id	500 Server error	User registration error
		400 Bad request	Malformed data supplied

User Management

API		Endpoint	
Get existing User Details Update existing User Details		GET /user PUT /user	
Security:		Requires end-user authentication token	
Inputs		Outputs	
JSON data: User Object	User Data to be updated	200 OK 200 OK JSON Data	User Data OK
Headers		Errors	
app-id	The unique application id	500 Server error	Internal Server Error
		400 Bad request	Malformed data supplied
		401 Not Authorised	If the certificate presented does not entitle the caller to execute this API
		404 No Data Found	No Data Found

User Login

API		Endpoint	
Login existing User		GET /user/login	
Inputs		Outputs	
JSON data: User Login Object	User Login Data	200 OK X-RTP-Access-Token (Res: headers)	Access Token to be send in further user API calls
Headers		Errors	
app-id	The unique application id	500 Server error	Internal Server Error
		400 Bad request	Malformed data supplied
		404 No Data Found	No Data Found

User Logout

API		Endpoint	
Logout an existing logged in User		PUT /user/logout?user=[user-id]	
Inputs		Outputs	
Uri: query	User id of the user	200 OK JSON Data	Message to user
Headers		Errors	
app-id	The unique application id	500 Server error	Internal Server Error
		400 Bad request	Malformed data supplied
		404 No Data Found	No Data Found

User Lookup

API		Endpoint	
Get existing User alias		GET /user/lookup?mobileNumber=[mobileNumber]|email=[email]	
Security:		Requires end-user authentication token	
Inputs		Outputs	
Uri: query - mobileNumber Uri: query - email	Mobile Number of the user Email of the User	200 OK JSON Data	PID of the searched user
Headers		Errors	
app-id	The unique application id	500 Server error	Internal Server Error
		400 Bad request	Malformed data supplied
		401 Not Authorised	If the certificate presented does not entitle the caller to execute this API
		404 No Data Found	No Data Found

Blocked User Management

API		Endpoint	
Get all blocked users Unblock a user		GET /user/blocked DELETE /user/{PID}/blocked	
Security:		Requires end-user authentication token	
Inputs		Outputs	
Uri: path - PID	PID of the user to unblock	200 OK JSON Data 200 OK JSON Data	Array of blocked PIDs PID of the unblocked user
Headers		Errors	
app-id	The unique application id	500 Server error	Internal Server Error
		400 Bad request	Malformed data supplied
		401 Not Authorised	If the certificate presented does not entitle the caller to execute this API
		404 No Data Found	No Data Found

Message transfer APIs (repo to repo)

All repo-to-repo APIs are secured via TLS and authenticated via TLS mutual authentication.

Send Message

API		Endpoint	
Send a message		POST /repo/{RQID}/messages	
Inputs		Outputs	
JSON data: A message Object to send	Message data	200 OK JSON Data	Message accepted for delivery. JSON data: RQID, Message ID
Uri: path - RQID	Request Id		
Headers		Errors	
repo-id	Sender repository id	400 Bad request	Malformed data supplied
		401 Not Authorised	If the certificate presented does not entitle the caller to execute this API

Process a Request

API		Endpoint	
Process a request		POST /repo/{RQID}/process	
Inputs		Outputs	
JSON data: A message Object to send	Message data	200 OK JSON Data	Message processed.
Uri: path - RQID	Request Id		
Headers		Errors	
repo-id	Sender repository id	400 Bad request	Malformed data supplied
profile	Always “RTP”	500 Server error	RTP Process error

Instrumentation, monitoring and QOS data APIs

Active Users

API		Endpoint	
Total active users		GET /qos/users	
Inputs		Outputs	
		200 OK JSON Data	Users data count provided, grouped by registered applications in the repository
Headers		Errors	
		500 Server error	Error in getting data
		404 Not Found	No data found

Threads (Requests)

API		Endpoint	
Total Threads (RTPs)		GET /qos/threads	
Inputs		Outputs	
		200 OK JSON Data	Threads' (RTPs) data count provided, for the Repository grouped by thread state - 'Active', 'Overdue', 'Completed'
Headers		Errors	
		500 Server error	Error in getting data
		404 Not Found	No data found

Messages Processed

API		Endpoint	
Total Messages Processed		GET /qos/messages	
Inputs		Outputs	
		200 OK JSON Data	Messages Processed data count provided, including status of messages for the Repository grouped by type of messages – 'sent', 'received', 'failed'
Headers		Errors	
		500 Server error	Error in getting data
		404 Not Found	No data found

Transactions (Money Processed)

API		Endpoint	
Total Transactions Processed		GET /qos/transactions	
Inputs		Outputs	
		200 OK JSON Data	Transactions Processed amount provided, including type of transaction for the Repository grouped by type – 'processed', 'requested'
Headers		Errors	
		500 Server error	Error in getting data
		404 Not Found	No data found

Data model

This section presents preliminary data model for the repository. Compliance of the Repository implementation is based on API compliance, and any data model that supports the APIs would be appropriate. In fact, repository operator does not need to disclose their data model as long as APIs are compliant.

Consumer IDV data

Table – IDV data		
Name	**Type**	**Description**
user_id	String	User ID in this repository. This is the first part of the PID.
auth_first_factor	String	First authentication factor. May be password hash.
auth_second_factor	String	Registered application name. May be memorable word.
Title	String	Person's title such as Mr, Ms, Mrs, Miss, Dr etc.
First name	String	First name
Middle name	String	Middle name
Surname	String	Surname
Address	Text	Address
Post code	String	Post code
Account number	Interger	Account number
Sort code	String	Sort code

Message data

Table – Message data		
Name	**Type**	**Description**
Message ID	64bit int	Unique identifier of the message within a Request to Pay thread
RQID	String	Unique ID of the Request to Pay. Can be viewed as a unique ID of the thread.
PID	String	Primary ID the message is addressed to.
Sender PID	String	Primary ID of the message sender.

In addition, the Repository operator would store a variety of transient data such as a token store in support of the oAuth2 implementation, or caches of TPP certificates, TPP API endpoints and certificates. This is implementation-specific transient data and is not detailed here.

7.4.3 Applications

Overview

Request to Pay Applications provide an interface for interacting with the service. Applications must be accredited to become part of the Request to Pay system.

Applications interact with repositories using repository APIs. In terms of interactions defined in the "standard" repository APIs, all accredited applications must be equally serviced by repositories. Where repositories publish extended set of APIs, applications may use this extended set based on separate agreement, provisioning and governance between applications and repositories.

There are several types of applications that are envisioned as part of the service, primarily distinguished based on audience or the type of organisation operating the application.

- Consumer applications
  - PISP / Fintech / innovator applications
  - Bank operated applications
- Business applications
  - Third-party request to pay business applications
  - Biller operated
  - Bank operated applications

Applications are core part of the ecosystem model presented in Figure 3 – Request to Pay ecosystem, and present broad competitive space for a variety of operators and integrators.

Functional description

Application functionality is described in detail in End-User requirements documentation, as well as previous study [Ref Z] of user experience and interaction design. Further, applications are subject to a range of requirements defined in chapters 5 and 6.

Application can connect to one or more of the user's message boxes (hosted on one or more repositories). After authenticating user with the respective repository for each message box, the application presents an interface to the user to view each request and associated messages.

A colour scheme may be used by the application to indicate status of the request – e.g. pending (amber), overdue (red), paid (green or grey). Similar visual cues should indicate whether payee is verified via Account Name Verification Service or Confirmation of Payee etc.

Where applicable, the application must provide the user with actions to take in response to a request. Actions include – Pay all, Pay partial, Decline request, Request Payment Period Extension for each request.

Applications must include a facility to initiate payments. Regardless of the payment mechanism, the application must:

1. Make best effort to only send "Pay all" and "Pay partial" messages if the electronic payment was successful (e.g. for non-immediate payments, initiation and all checks are performed without failure)
2. Must not send any messages to Biller in case payment fails.

8 Request to Pay Demonstrator

The objective of the Request to Pay demonstrator is to validate:

- Architecture approach
- Messaging patterns
- User experience

It is therefore expected that interactions between components are as close to reality as possible, while actual component implementations are less relevant.

To accelerate implementation, a following set of tools has been proposed.

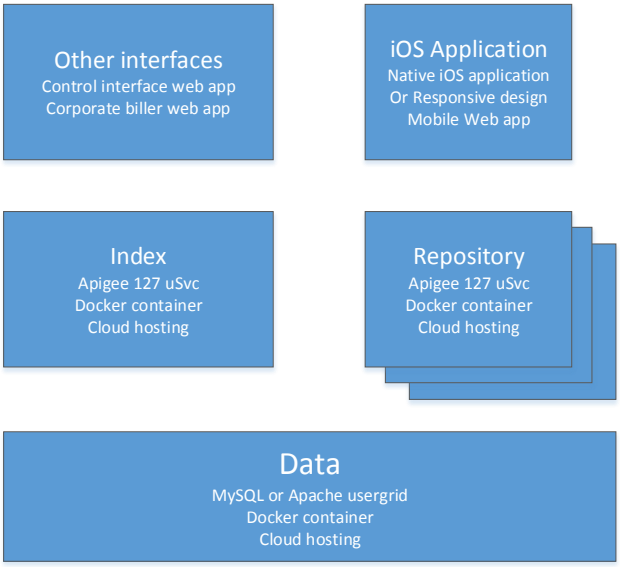


Figure 19 Preliminary components and implementation technologies for RtP demonstrator

In order to avoid complex licensing terms, only non-commercial open source components are to be used. However, this will reduce the out-of-the-box functionality especially in the API Platform part - features from commercial API platforms such as traffic management, oAuth2 implementation and analytics may not be available out of the box.

Components are to be hosted in the cloud (AWS or GCP), and each of the infrastructure components (repository, index) is to be encapsulated in a deployable Docker image.

TLS Certificates will be manually created for imaginary TPPs operating repositories or Applications. TLS termination and certificate checks may be implemented in a custom component or using native cloud features.

Depending on use cases, the demonstrator will need to be able to operate a minimum of three repositories.

Application implementation may be either iOS native or a single-page Web Application with responsive design (rendering on mobile and desktop browsers).

Should time allow, a demonstration of command and control interface may be designed with mock data.

Should time allow, a demonstration of what a corporate Biller application may look like (e.g. for bulk requests) may be designed. Should time allow, this design may connect to Repository demonstrators.

9 API Design Guidelines

9.1 Overview

An API (Application Programing Interface) defines the bridge for communication between an application and the backend service. These interfaces provide a pre-determined set of communication "methods" along with input/output parameters for sending and receiving messages for communicating with an application over the network.

RESTful web APIs, provide a simplified approach for communication using the http protocol and REST API design principles. In this document, we would look at the guidelines and design standards to be followed for building a RESTful API. These standards will help to design APIs that are intuitive and can be easily understood and consumed by developers consuming them.

9.2 RESTful API Design Guidelines

9.2.1 API URL Naming Conventions

API URLs must follow a naming conventions that meets the following criteria:

- It must provide the address or location of the server hosting the API
- It must specify the service name of the API
- It must specify the version of the API service
- It must specify the resource on which the service operation is being performed
- It must follow a hierarchical approach for traversing nested sub-resources

Following convention must be followed for forming the API URL:

http[s]://[<server-name>]/api/[<service-name>]

where

- **server-name** - The hostname or IP given to the installed web server that is running the API.
- **service-name** - The API name of the service you want to access

Example: '`https://api.<companyname>.com/api/amenities`'

9.2.2 URI Versioning

Versioning is an important aspect of API design. Every API must have a version number associated with it. Hence, the URL must specify a version number to identify the version of the API. There are various approaches for versioning. But to keep things simple, it is recommended to use an integer number prefixed with a 'v' to denote the version of the API.

Example: '`https://api.<companyname>.com/api/amenities/`**`v1`**'

9.2.3 URI Format

A URI consists of segments separated by forward slashes ('/'). Each segment must identify a resource. If a resource has sub-resources, the URI must specify the path to the sub-resource in a hierarchical manner as follows:

/resource/path/to/sub-resource

Example: '`https://api.<companyname>.com/api/amenities/v1/`**`flights/UA881/cabin`**

9.2.4 Resource Naming Convention

Every resource of an RESTful API must have a meaningful name to identify itself. It is recommended to name a resource using noun as opposed to verb or action. The URI of the resource must refer to a thing rather than an action. Also, CRUD function names should not be used in the resource names. Hence use of resource names like 'getflights' to retrieve information about flights must be avoided.

A collection of resources can be named using a plural noun. Eg.

`/api/amenities/v1/`**`flights`**

Following are some of the recommended naming conventions for URI Path for a RESTful API

- Name collection resource with *plural noun*: Eg.
  `http://api.foo.com/api/amenities/v1/flights`
- Name singular resource with *singular noun* : Eg.
  `http://api.foo.com/api/ameneties/v1/flights/UA123`
- Name a controller resource using a *verb* : Eg.
  `http://api.foo.com/api/amenities/v1/flights/UA123/book`
- Avoid using CRUD operation names in the URIs. For example, do not use URIs like
  `http://api.foo.com/api/amenities/v1/getflights`
- Use lowercase for naming URIs. Avoid mixed and upper cases in URIs. Mixed case is harder to type in and read.
- Use hyphen instead of space or underline. They are aesthetic and easier to read. Spaces in URL will get transformed into URL encoded %20s, degrading readability further. For example, use URIs like http://api.foo.com/api/about-us
- Avoid using characters that require URL encoding. Eg. Spaces

9.2.5 Modelling Resources and Sub-Resources

A resource can be a single instance of an object or a collection of objects. For example, a collection of flights can be represented by the object '/flights'. Again, a single flight within this collection can be identified by the flight number as '/flights/UA123'. There can be further objects that can be related to each other either as parent child or in some other ways. For example, a flight may have different amenities. Hence 'amenities' can be a sub-resource of a parent 'flight' resource and can be related as follows: /flights/UA123/amenities

The following approach should be followed for designing URI path with resources and related sub-resources:

URL	Description
/servicename/v1	This is the entry point for the API
/servicename/v1/{ResColName}	Resource name of a top-level collection
/servicename/v1/{ResColName}/{ResId}	A resource instance within the collection of resource

/servicename/v1/{ResColName}/{ResId}/{SubResColName}	A sub-resource collection under the resource ResId
/servicename/v1/{ResColName}/{ResId}/{SubResColName}/{SubRedId}	SubRedId inside the collection of SubResource

Table 5 URL and description

9.2.6 HTTP Verbs

After identification of the resources, the next question is about the action to be performed on these resources. A HTTP verb is normally used to specify the action to be performed. It forms an important part of RESTful API design. The primary and most commonly used HTTP verb are POST, GET, PUT and DELETE. These verbs help to perform the CRUD operations on the resource. As a guideline, these verbs should be used as follows:

- GET – Used to retrieve or read the information about the requested resource entity identified by the request-URI
- POST – Used to create a new resource, which is subordinate to the parent resource identified by the request URI
- PUT – Used to update an existing resource entity identified by the request URI
- DELETE – Used to delete the resource represented by the request URI

Other HTTP verbs like PATCH, OPTIONS and HEAD can be used for specific usage requirements

9.2.7 API Payload Format

The API request or response body content is referred to as the API payload. There are many options for exposing the API payload. But in the RESTful world there are 2 well adopted formats – viz. JSON and XML. JSON is the preferred of the two due to the following benefits that it offers over XML:

- JSON is a more compact format, meaning it weighs far less on the wire than the more verbose XML. It is a good benefit for mobile devices with a limited bandwidth to save costs and improve loading speed.
- JSON parsing is generally faster than XML parsing.
- JSON is easier to work with in some languages (such as javascript, python, and php)
- Formatted JSON is generally easier to read than formatted XML.
- JSON specifies how to represent complex datatypes, there is no single best way to represent a data structure in XML.

Due to these advantages, JSON data format should be preferred and default format used to specify the RESTful API payload. XML output can be supported based on the 'Accept' header specified in the request.

9.2.8 API Headers

One area of REST API design that warrants attention is the use of "Media Types", which are also known as either MIME Types or Content Types. Media types have the following syntax:

```
type "/" subtype *( ";" parameter )
```

REST APIs typically work with media types that fall under the "application" type. Note that parameters may follow the type/subtype in the form of attribute=value pairs that are separated by a leading semi-colon (;) character. HTTP/1.1 uses media types in the values of the 'Accept' and 'Content-Type' headers. As shown in the example below, client applications can convey their preference for a response body's media type using HTTP/1.1's 'Accept' request header.

```
Accept: application/json,application/xml;q=0.9,text/html;q=0.8,*/*;q=0.7
```

In the 'Content-Type' header of an HTTP/1.1 request or response, a media type reference indicates the "type" associated with the message body's byte sequence. The example below demonstrates a Content-Type header value that references a media type with a "charset" parameter:

```
Content-type: application/json; charset=ISO-8859-4
```

REST APIs use either the "application/json" or the "application/xml" media type in the 'Content-Type' header of an HTTP/1.1 request or response.

9.3 Exception Handling

9.3.1 Error Handling

Communicating error information properly to the API consumer is critical for the success of the REST API. API consumers and app developers using APIs learn to write code through errors. Well defined error messages are helpful for troubleshooting and resolving issues after the applications built using the APIs are in the hands of the end users.

RESTful APIs must communicate error information using proper HTTP response status code. There are different types of HTTP response status codes to communicate the different success and error information as follows:

- *2xx: Success* – Used to communicate that the request from the client was successfully received, understood, and accepted
- *3xx: Redirection* – Used to communicate that additional action needs to be taken by the user agent like browser in order to fulfil the request
- *4xx: Client Error* - Used to indicate errors caused by the client
- *5xx: Server Error* – Used to indicate that server is aware that an error occurred while processing the request and cannot process it further

Http Error Response The following table summarizes the HTTP error response codes that may be returned by the API under different error scenarios.

400	Bad Request	Indicates that the request had some mal-formed syntax error due to which it could not be understood by the server. Probable reason could be missing mandatory parameters or syntax error
401	Unauthorized	Indicates that the request could not be authorized possibly due to missing or incorrect authentication token information
403	Forbidden	Indicates that the request was understood by the server but it could not be processed due to some policy violation or the client does not have access to the requested resource

404	Not Found	Indicates that the server did not find anything matching the Request-URI
405	Method Not Allowed	Indicates that the method specified in the Request-Line is not allowed for the resource identified by the Request-URI
409	Conflict	Indicates that the request could not be processed due to a conflict with the current state of the resource
414	Request URI Too Long	Indicates that the Request URI length is longer than the allowed limit for the sever
415	Unsupported Media Type	Indicates that the request format is not supported by the server
429	Too Many Request	Indicates that the client has submitted the request too often and needs to slow down
500	Internal Server Error	Indicates that the request could not be processed due to an unexpected error in the server.
501	Not Implemented	Indicates that the server does not support the functionality required to fulfil the request
502	Bad Gateway	Indicates that the server while acting as a gateway or proxy received an invalid response from the backend server
503	Service Unavailable	Indicates that the server is currently unable to process the request due to temporary overloading or maintenance of the server. Trying the request at a later point of time might result in success
504	Gateway Timeout	Indicates that the server while active as a gateway or proxy did not receive a timely response from the backend server

Along with the Http error status code, the response message must also provide additional information to clarify the error. The following payload format can be used to communicate additional information about the error:

```
{
      “status” : {status [optional]},
      “code” : {code [optional]}    ,
      “message”: “Error message”,
      "errors": [optional]
      [
            { "code": {error code},
            "message": "Error message"
            }
]
```

Extra fields can be added as needed. The errors array is an optional attribute, which will often be used when the service captures multiple errors to return to the consumer.

Example for error message payload is as follows:

```
{
      “status” : 400,
```

```
      “code” : 40010,
      “message”: “SMS message body is not specified”,
      "errors": [
                  { "code": 1, "message": "SMS message body is required"
            } ,
                  { "code": 2, "message": "SMS recipient is required" }
            ]
}
```

Appendix 1: Request End State

Further a need was identified to understand when a request is "closed", under which conditions and what end states of the request are possible.

End states

General approach is that there are two way to close the request:

- Payment made in full
- Request declined
- Payment period ends

Following end-states are currently under consideration:

- Paid fully
  - Description: The request is paid by the Payer
  - Typical cases: The full amount is paid before the end of the payment period (either with a single Pay-all payment or multiple Pay-partial payments)

- Paid partially
  - Description: The request is partially paid, but the payment period has expired
  - Typical cases: While a partial payment has been made, payment period has ended without full payment.

- Not paid
  - Description: The payment period has expired without a payment being made
  - Typical cases: Payer ignored the request and has not made a payment.

- Rejected
  - Description: The Payer has rejected the request
  - Typical cases: Payer did not recognise the charge or has made the payment outside the RtP system.

It is worth noting that requests will only ever be soft-closed. That is to say, no data is deleted and no irreversible action is performed on the request; the "message thread" remains, and both biller and payer are able to post new messages (e.g. contact biller / contact payer would still be possible).

In effect, the biller and payer applications would determine the status of a request by analysing messages in the thread.

Payer end state (without biller reconciliation)

From the Payer's perspective, requests are "closed" based on Payer actions (e.g. pay, decline etc.) There are several good reasons for this.

A. Payer applications would only send a message when a payment (to the account

specified by the Biller) is successful.

B. Request to Pay ensures that Payee account is part of the request, and is validated using Confirmation of Payee, and therefore it is very unlikely that the payment would be made to wrong account.

If a biller determines that the payment has not been made (despite RtP message), biller should have a facility to reopen the request via additional Request to Pay message. This is preferred to contacting the payer out-of-band or activating delinquency process.

Biller end state (with biller reconciliation)

As discussed, biller end state is determined by biller's accounting systems.

If we take the position that biller acknowledgement is required to "close" the Request to Pay for biller – for example after account reconciliation on biller side - biller application would be required to detect payment of a particular request and send a message to acknowledge payment.

In this case, a paid request would remain open (or perhaps somewhere between "open" and "closed") until acknowledgement is received, even if payment period ends. This also has ramifications on the "Paid partially" end state – where the request would remain open until biller confirmation of the partial payment.

Main benefit of this approach is that biller state is accounted for in "closing" the request. However, this also has some drawbacks.

- Should this interim state be exposed to end users (payer in this case) - this may seem odd to the end-users who are not used to having visibility of biller settlement. Could be particularly difficult in case settlement takes days or weeks.
- This may increase complexity of biller applications as they must send messages to acknowledge payment for each payment on each request.
- This would increase the complexity of entering end-states – as reconciliation may take any amount of time, closing of requests may need to be delayed beyond end-date (also applies to partial payments where we may need to wait for multiple settlement messages).

Appendix 2: Functional Requirement Summary

In this section, we enumerate functional requirements established to date. This section is work in progress.

Category	Number	Functional requirement
General	G1	Request to Pay implementations must be invariant to payment methods.
TPP	T2	Allow Third-Party operators to operate end-user applications
TPP	T3	Allow Third-Party operators to operate message repositories
TPP	T4	Provide a central Third-Party registry that provides standard PKI infrastructure, including issuing certificates and real-time certificate revocation checks.
End-user	E1	Allow end-users to register with Third-Party providers
Repository	R1	Support transporting messages between Payer and Payee who may have different service providers (third party operators).
Repository	R2	Support any number of messages associated with a specific request to pay, and easy retrieval of all message associate with a specific Request to Pay.
General	G2	Support message attachments so that a bill may be attached to a request, or a receipt may be attached to responses.
General	G3	Support validation of message format and type in the context of original request, to provide assurance message can be understood by the other party
Index	I1	Support validation of Payee details by Payer – e.g. via Confirmation of Payee
General	G4	DUPLICATE of R1 - Support transporting messages between Payer and Payee who may have different service providers.
Index	I2	Support mutual authentication between any two third-party providers for each transaction between them.
Index	I3	Support mutual authentication between central infrastructure and third party providers for each transaction.
Index	I4	Support real-time checks of Third-Party provider identity
General	G5	Support message authentication codes that will prevent tampering with a message in the delivery path.
General	G6	Each participant in message delivery (repository, application) must provide confirmation of message receipt and notification of failure to deliver a message.
General	G7	Each participant in message delivery (repository, application) must provide a facility to retrieve all messages associate with a specific Request to Pay
General	G8	Support uniquely identifying end-users based on a PID
General	G9	Support email and mobile phone number as PID aliases
General	G10	Support end users who do not have a bank account as Payers
Application	A1	Third-Party Application providers must support at least one electronic payment method
General	G11	The service implementation must not exclude cash payments, however Third Party provider may choose not to provide this facility
General	G12	The service implementation must not exclude end-users who do not have smart devices
General	G13	The service implementation must not exclude end-users who do not have internet connectivity
Application	A2	Request to Pay may only be generated with a full PID of the Payee
Application	A3	Request to Pay may only be generated if a Payee has a bank account on file
General	G14	Associating a Payee with a bank account must involve Confirmation of Payee check
Application	A4	Payer must clearly see the result of Confirmation of Payee query on the Payer
Index	I5	When looking up an alias, Index must not disclose PID but must rather return only the Repository endpoint for the given alias.

Table 6 Functional requirements