PISP/Docs

Supporting documentation for the PISP implementation.

Overview

Tools

To update the sequence diagrams in `./docs/out`, ensure you have the PlantUML vscode plugin installed.

1. `CMD + Shift + P`
2. Select `PlantUML: Export Workspace Diagrams`
3. Wait for the export to complete, and commit the changes

```ts
export default {
  name: 'MyComponent',
  // ...
}
```

> **TIP**
>
> This is a tip

> **WARNING**

DANGER

This is a dangerous warning

▶ DETAILS

Edit this page on GitHub

Design Decisions

> For now, let's place the decisions inline for ease of reference, but we may want each decision to have it's own `.md` file in the future.

Outstanding Questions

- Q: How does the switch *know* to send a callback to the PISP after a sucessful transfer?
  - Refer to #42
- Q: Will we make a new `thirdparty-scheme-adapter` to handle thirdparty requests?
  - Signs point to yes at the moment, but the challenge is how to divide between the existing `sdk-scheme-adapter` and a new `thirdparty-scheme-adapter`

Decisions Made

How does the switch determine whether or not a DFSP is using their own FIDO service? Do we want to use the ALS or some other method?

We will use the ALS to record the auth service for a given participant

For example, to find the Auth service for `dfspa`, a participant can call `GET /participants/AUTHSERVICE/dfspa`

Example 1. internal auth-service

request

```
GET /participants/AUTHSERVICE/dfspa
```

response

```
{
  "fspId": "switch"
}
```

Example 2. dfsp's own auth service

request

```
GET /participants/AUTHSERVICE/dfspb
```

response

```
{
  "fspId": "dfspb"
}
```

Will the `pisp-demo-server` use the `sdk-scheme-adapter` / `thirdparty-scheme-adapter` ? Or will it speak native async mojaloop?

It will speak native async mojaloop, so will not be using any adapter. It will however use the `sdk-standard-components` which is currently being updated for

For now, we are adding PISP functionality to the sdk-scheme-adapter primarily because the mojaloop-simulator requires it for our end to end tests.

Which api should the DFSP need to implement for PISP functionaliy? We have a few options:

1. Add the DFSP changes to the existing FSPIOP-API
2. Add the DFSP changes to the new `thirdparty-api`
3. Divide the `thirdparty-api` into 2 parts:
   - `thirdparty-pisp-api` for the PISP to implement
   - `thirdparty-dfsp-api` for the DFSP to implement

A: Option 3: We are going to divide the new thirdparty-api into 2 parts.

Refer to DA issue #47 for more information about this decision.

How should we implement the changes required for the PISP role?

- Should we extend the existing APIs or should we create one or more new APIs to manage the specialised PISP interactions?

There is still the outstanding question for the DFSP side of the equation (See above)

Should the `mojaloop/auth-service` API be Sync or Async?

It will be Async

What is the challenge that is being signed during the transfer flow?

The condition from the `QuoteResponse` object should be signed by the PISP app running on the user's device.

PISP transfer initiation resource

For the PISP role, we will add an additional resource to the API to initiate a 3rd party payment

- currently we are planning on calling it `/thirdPartyRequests/transfer`. This may change.
- For our current implementation work however, we can still use the existing `/transactionRequests` resource.

Handling of an `/authorization` for external FIDO

In the case where a DFSP wishes to bring their own FIDO Service instead of using the FIDO service that is a part of the hub (`mojaloop/auth-service`), we want to pass on the `PUT /authorization/{id}` (see `AG-26` in the Transfers E2E flow) directly to the DFSP for their own FIDO Service to verify that the public key matches the signed challenge.

This means that we don't need to unpack the `authorization` object, nor do we need to design a new API Endpoint for external auth services.

This does, however, require some thinking about the error cases when verifying the signed challenge. Here is what we have proposed:

- internal (Mojaloop-hosted) fido: failure case for FIDO -> forward error to DFSP in `PUT /authorizations/{id}/error`
- external fido: forward whole `PUT /authorizations/{id}` request to the DFSP to take care

Mojaloop PISP

Bringing 3rd Party Payment Initiation to Mojaloop

Get Started →

Simplicity First

Minimal setup with markdown-centered project structure helps you focus on writing.

Vue-Powered

Enjoy the dev experience of Vue + webpack, use Vue components in markdown, and develop custom themes with Vue.

Performant

VuePress generates pre-rendered static HTML for each page, and runs as an SPA once a page is loaded.

Error Codes

This document is suggesting that a high level error category 6xxx(can be another number if 6xxx is reserved) be used to represent errors that are returned to a third party such as PISP's and AISP's. These are error codes the Switch would send back to a PISP.

- **Third Party Error -- 60*xx***

Error Code	Name	Description	/parties	/thirdPartyRequest	/consentRequests
6000	Third party error	Generic third party error.	X	X	X
6001	Third party request error	Third party request failed.	X	X	X

- **Permission Error -- 61*xx***

Error Code	Name	Description	/parties	/thirdPartyRequest	/consentRe
6100	Authentication rejection	Generic authentication rejection			X
6101	Unsupported authentication channel	Authentication request is attempting to authorize with an authentication channel that the DFSP does not support. Example Web, OTP.			X
6102	Unsupported scopes were requested	Authentication request is attempting to get authorization for scopes that the DFSP doesn't allow/support			X

6103	Consent not given	DFSP denies user gave consent or DFSP says user has revoked consent.			X
6104	Consent not valid	DFSP denies user has valid consent with correct credentials.		X	X
6105	Thirdparty request rejection	DFSP catch all error used when it rejects a thirdparty request.		X	X

- **Validation Error** -- 62*xx*

Error Code	Name	Description	/parties	/thirdPartyRequest	/consentRe
6200	Invalid signed challenge	PISP server/DFSP receives signed challenge that is invalid.			X
6201	Maximum authorization retires reached	PISP server has reached maximum number of authorizations.			X
6202	Missing authentication credential	Payload received with missing authentication credential.			
6203	Invalid authentication token	DFSP receives invalid authentication token from PISP.			
6204	OTP is incorrect	One time password is incorrect.			

6205	Mismatched thirdparty ID	Thirdparty ID doesn't match corresponding thirdparty request.			

Edit this page on GitHub

branch strategy:

1. When we are creating a new branch we are using `pisp/` prefix in branch name.
2. We have `pisp/master` leading branch to keep our final feature candidate to be merged with `master`
3. Newly created PR for every WIP branch should have `pisp/master` set as target
4. When the new version of `master` is published (by other team for example), we should propagate all changes via merge with `pisp/master`

PISP changes in other Repos :

1. POST /authorizations Swagger changes #269
   - mojaloop-specification
   - transaction-requests-service

Identifiers

> ***Note:*** *This isn't the final destination for this documentation, I just figured this repo would be a good place to add any new docs. If you think it belongs elsewhere, please suggest a new location.*

Purpose:

Within the PISP world we are designing, we need a means to associate a PISP's view of a (user + device + account) with a DFSP's view of a (Party + Account), and be able to share such an association with other participants (DFSPs) in the Mojaloop Ecosystem.

For example:

1. Ayeesha holds 2 accounts with `DFSP A`, a Savings Account and Chequing Account
2. She also has a PISP account with `PISP X`, and has associated her Chequing Account with `DFSP A` with the PISP's app on her mobile device
3. Ayeesha wishes to send to Bravesh, who holds an account with `DFSP B`
4. Ayeesha wants to initiate the payment from the convenience of her PISP app (this flow is covered elsewhere)
5. Since she holds more than 1 account with `DFSP A`, the PISP needs a method to "tell" `DFSP A` which account Ayeesha wishes to use, and to debit.

This document aims to define the relationship between a party's account with a DFSP and a (Device + User + Account) registered with a PISP. Such a relationship needs to be:

1. **routeable:** Today, the Mojaloop ALS (account lookup service) is for looking up *parties* and their associated *DFSPs* and not specific *accounts*. For the sake of the PISP proposal, this means we need a new way to identify both a party and it's account which is held with a DFSP.
   - 1.1 This allows a PISP to "tell" a DFSP which account the user intended to send the funds from
2. **secure:** We don't want to ask DFSPs to disclose potentially sensitive information about a user and/or their account to the PISP if we don't have to.

Flows:

Basic Sequence Diagram

1. Ayeesha registers her chequing account with DFSP A to the PISP App

> ... Account Linking Steps here...

to Ayeesha's Chequing account

3. DFSP A informs the PISP Server of the UUID, and asks it to use that UUID to refer to Ayeesha's chequing account on the PISP End
4. Async? callback
5. Internal processing
6. DFSP A asks the switch to create a new `ASSOCIATION` for `1111-2222`
7. Async Callback
8. Switch asks the ALS to create a new key/value pair for `ASSOCIATION/1111-2222` + `DFSPA`
9. Async Callback
10. Switch calls back to DFSP A, saying that the association has been created
11. DFSP calls back to the PISP Server, informing it that the association has been created

Questions:

Q: Why can't the association between a PISP and DFSP Account be stored either with the DFSP or PISP? For example, the DFSP could give an account number to the PISP to store. Or the opposite, the PISP could generate their own association ID to give to the DFSP to be able to identify the sending account. A: A few reasons:

- There is no standard for accounts defined in Mojaloop. For the DFSP to give an 'account number' to the PISP, this would need to be defined, and would make the solution less generalizable accross a myriad of DFSPs
- Even if we did have a standard for 'account number' the DFSP could share the a PISP, do we really want to ask the DFSP to disclose this information? In some instances it might be ok from a security perspective, but once again, that is heavily dependent on the DFSP's own implementations, which we don't want to get involved with.
- The PISP could create an identifier and ask the DFSP to store this for them, but it seems to me that PISPs should be "read only" from DFSPs, and asking DFSPs to store a value on behalf of the PISPs would break the existing convention we have.

> Could someone please add to this answer? Or let me know if it even makes sense? I'm sure there's other reasons we don't want to do this.

Previous Discussion

> A summary of the existing discussion from our meeting + Slack.

London Meeting Notes

Account identifiers and Information

- Need more work on this, most focus thus far has been on sending, not receiving
- e.g. From Alice@DFSPA to Bob@DFSPB
- But DFSPA needs to know which account to debit
- Therefore need to assign identifiers
    - because we are “sending from the account”
- Identifiers act as a link between PISP to a single account at DFSP
- PISP should know about it...
- need to do GET parties *both* for Sending Party and Receiving Party
    - E.g. check the sending account is not frozen. Give the Sending DFSP a chance to say "Wait, no this acc
    - need to map between what the PISP knows with what it can Tell the other Mojaloop participants

Slack

Michael Richards:

Thoughts on identifiers:
As part of the association process, either the PISP or the DFSP creates a nonce value (e.g. a UUID) to identify the association with a particular account.

1. We add a new identifier type, say ASSOCIATION
2. The PISP and the DFSP share the newly created value
3. The DFSP registers that value with the ALS
4. Now, when the PISP wants to identify the DFSP associated with a particular association, it can simply ask the ALS: who owns this association value? and the ALS responds: the DFSP...

Lewis Daly:

Thanks Michael, can you please jog my memory about the problem being solved with this?

Sam:

@Lewis Daly this is a proposal I think, to provide for identifying & associating multiple accounts (understanding the possibility of having multiple accounts for a same ID). Right now, the API

matdehaast:
And also ensuring account data provided to PISP's aren't sensitive data and mapped internally

Michael Richards:

The way in which Mojaloop obtains the route to a DFSP is by using an identifier - a `MSISDN`, or `IBAN`, or whatever - to get an answer from the ALS: apply to this DFSP.

The DFSP then associates the identifier with an actual transaction account using its own internal processes which are undefined as far as Mojaloop is concerned. For associations between a PISP and a DFSP, we need to ensure that the identifier used by the PISP will always route to the associated account, so we want to make the identifier specific to that account. Which is what this solution is designed to achieve...

Lewis Daly:

Ah I see, thanks. And as Matt said, we also want to make sure a PISP doesn't know sensitive information about the account holder, right? Otherwise we could determine the UUID deterministically (ie. a hash value of the dfspId, pispId and account iban/msisdn)

Michael Richards:

Yes, indeed. You should keep arbitrary things arbitrary, in my view...

Contents

1. Linking

The goal of the linking process is to explain how users establish trust between all three interested parties:

1. User
2. DFSP where User has an account
3. PISP that User wants to rely on to initiate payments

NOTE: For now, we're focusing on the FIDO / Web authentication channel.

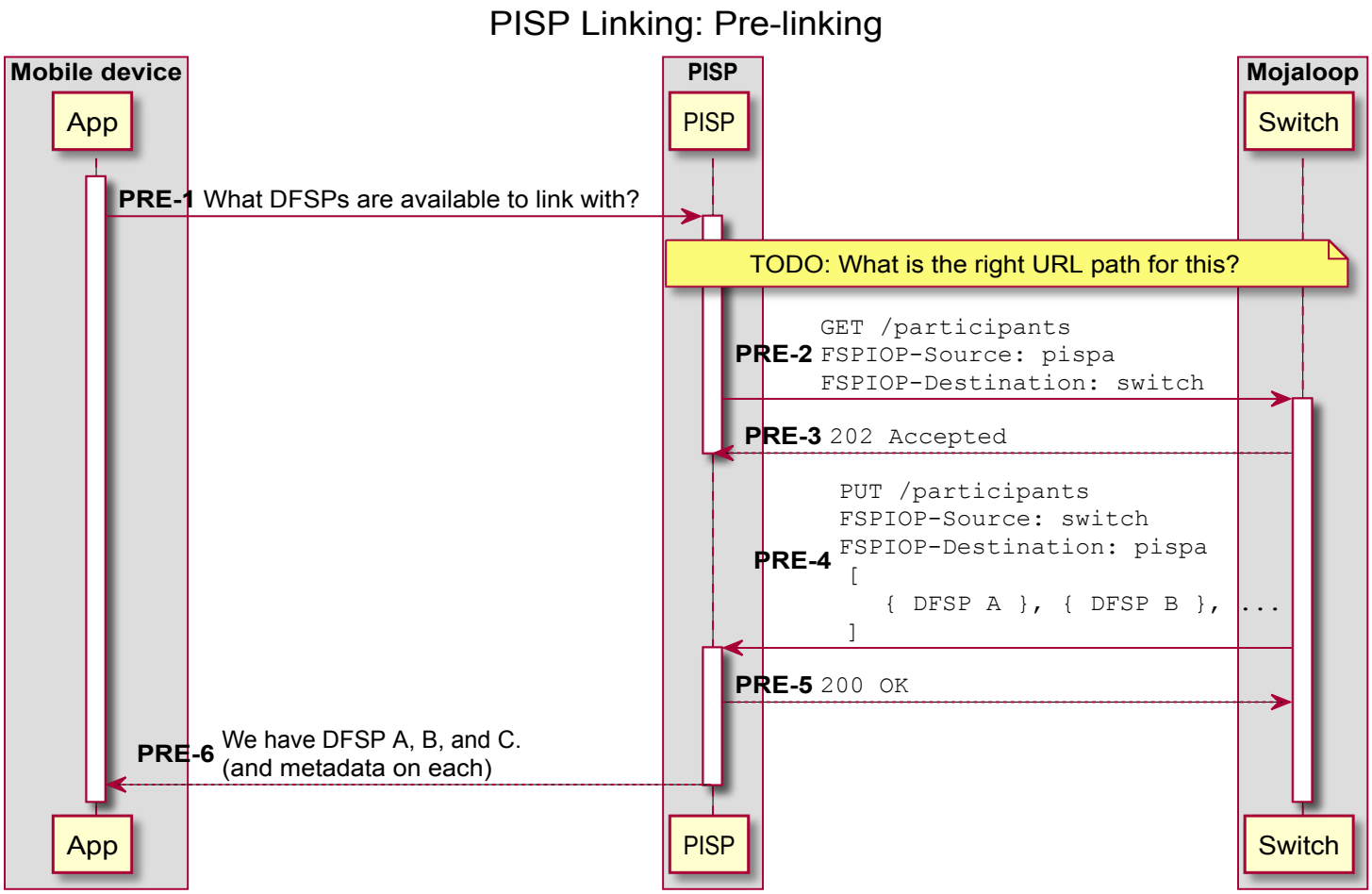
Linking is broken down into several separate phases:

1. **Pre-linking** In this phase, a PISP asks what DFSPs are available to link with.
2. **Request consent** In this phase, a PISP attempts to establish trust between the 3 parties.
3. **Authentication** In this phase, a User proves their identity to their DFSP.
4. **Grant consent** In this phase, a PISP proves to the DFSP that the User and PISP have established trust and, as a result, the DFSP confirms that mutual trust exists between the 3 parties.

1.1. Pre-linking

In this phase, a PISP Server needs to know what DFSPs are available to link with. This is *unlikely* to be done on-demand (e.g., when a User clicks "link" in the PISP mobile App), and far more likely to be done periodically and cached by the PISP Server. The reason for this is simply that new DFSPs don't typically join the Mojaloop network all that frequently, so calling this multiple times on the same day (or even the same month) would likely yield the same results.

The end-goal of this phase is for the PISP Server to have a final list of DFSPs available and any relevant metadata about those DFPSs that are necessary to begin the linking process.

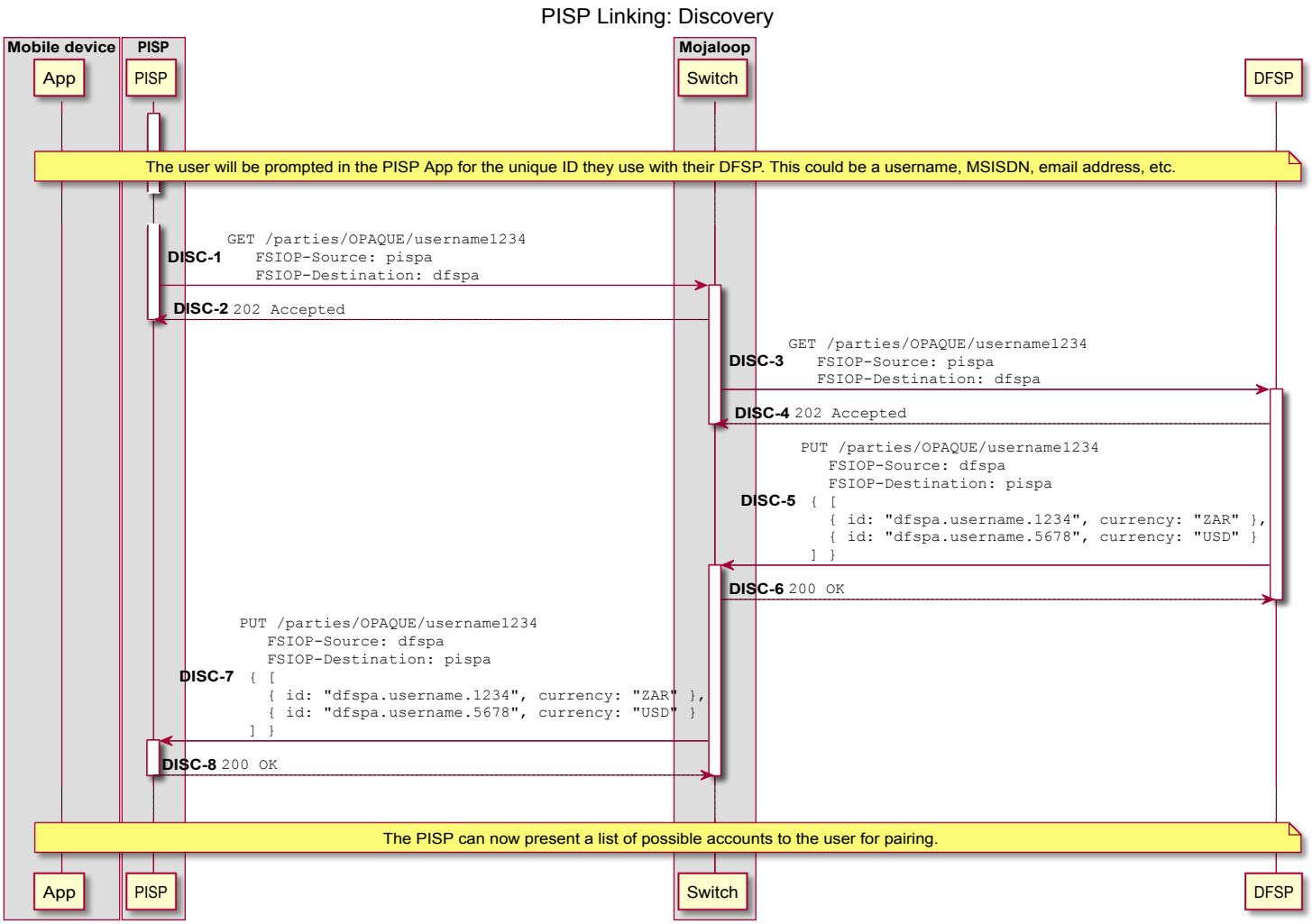


1.2. Discovery

In this phase, we ask the user for the identifier they use with the DFSP they intend to link with. This could be a username, MSISDN (phone number), or email address.

The result of this phase is a list of potential accounts available for linking. The user will then choose one or more of these source accounts and the PISP will provide these to the DFSP when requesting consent.

decide during the Authentication phase that they actually would like to link a different account than those chosen at the very beginning. This is perfectly acceptable and should be expected from time to time.



1.3. Request consent

In this phase, a PISP is asking a specific DFSP to start the process of establishing consent between three parties:

1. The PISP
2. The specified DFSP
3. A User that is presumed to be a customer of the DFSP in (2)

The PISPs request to establish consent must include a few important pieces of information:

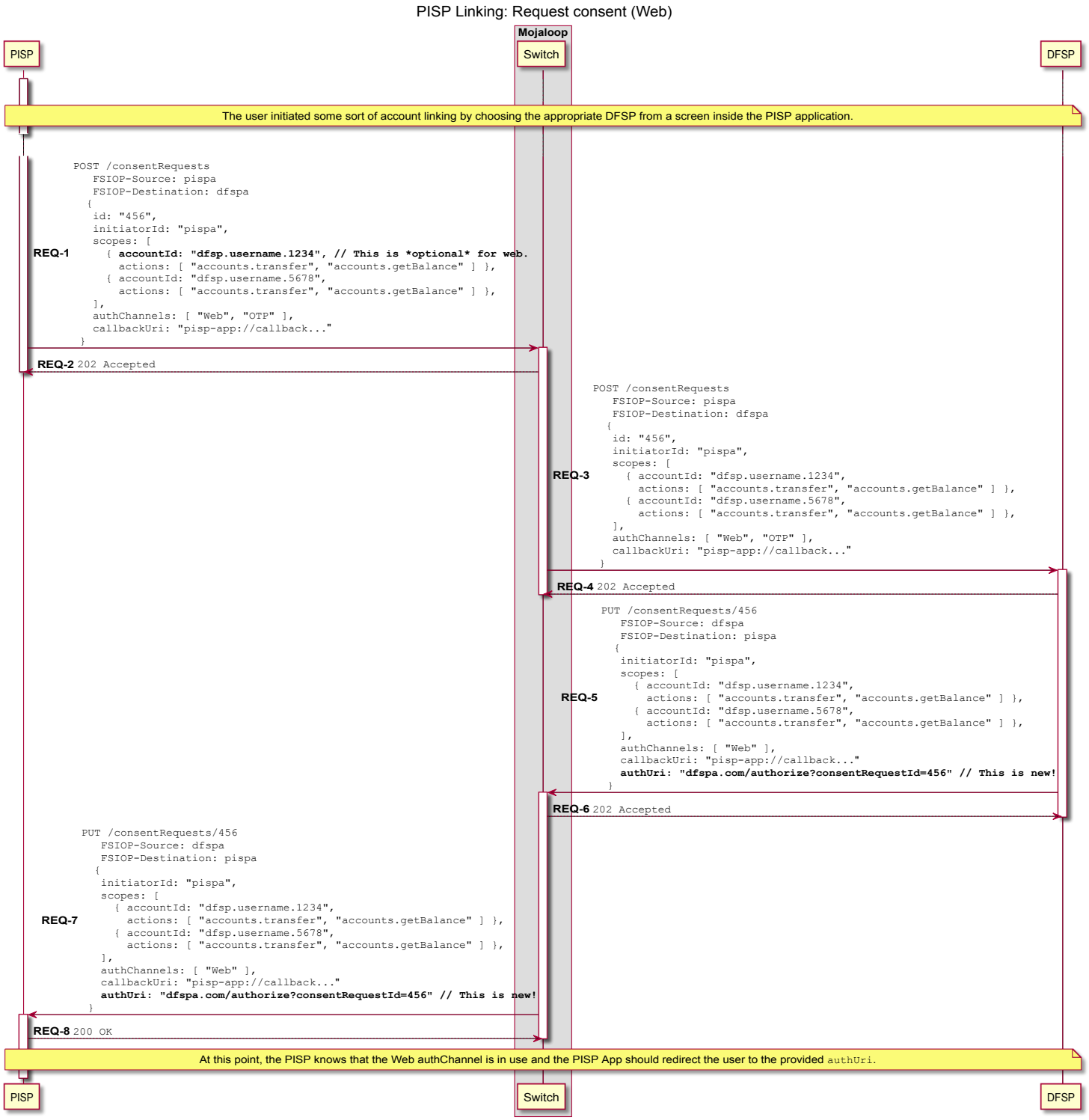
- The authentication channels that are acceptable to the User
- The scopes required as part of the consent (in this case, almost always just the ability to view a balance of a specific account and send funds from an account).

- A callback URI of where a user can be redirected with any extra information.

The end result of this phase depends on the authentication channel used:

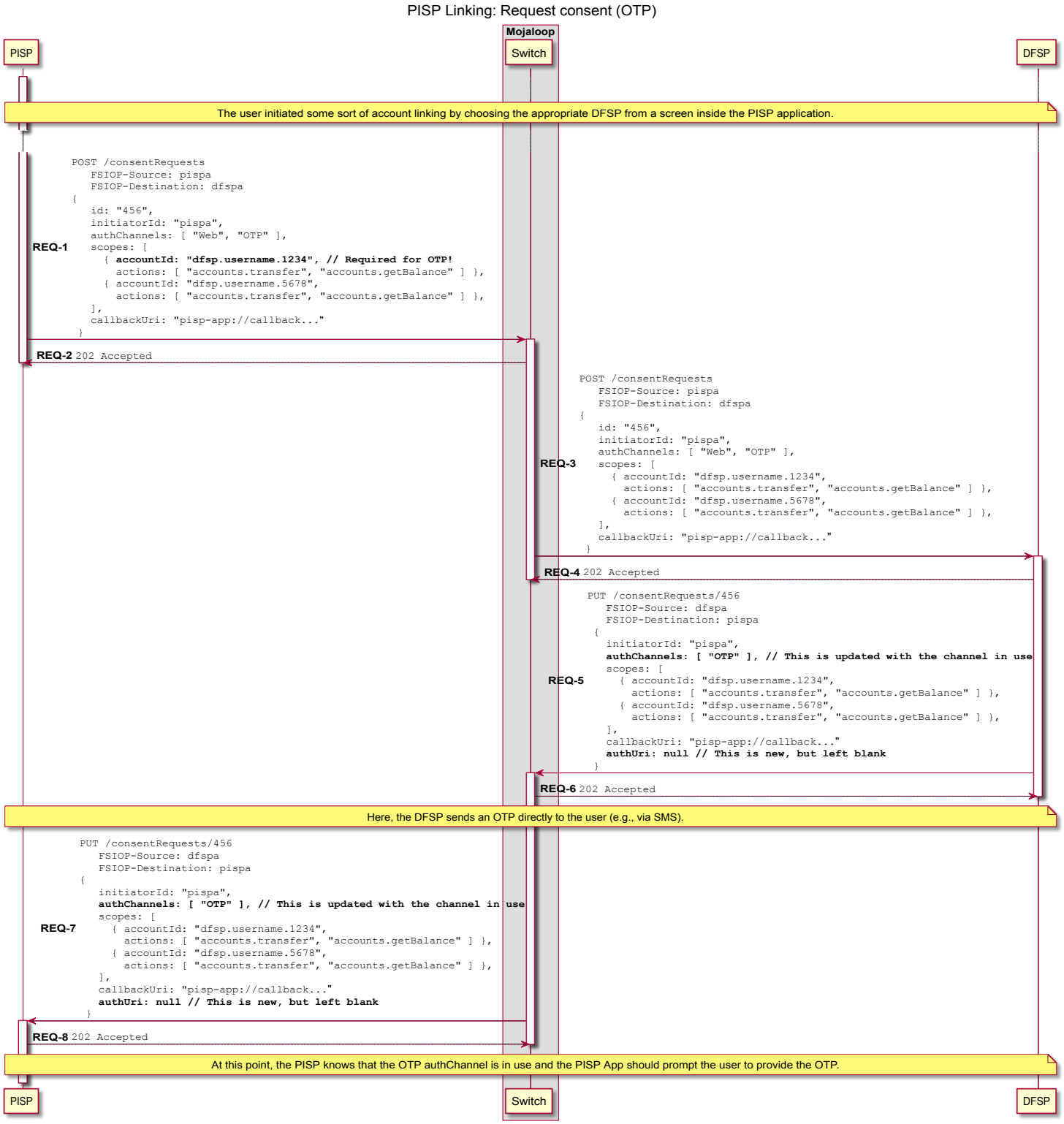
1.3.1. Web

In the web authentication channel, the result is the PISP being instructed on a specific URL where this supposed user should be redirected. This URL should be a place where the user can prove their identity (e.g., by logging in).



1.3.2. OTP / SMS

In the OTP authentication channel, the result is the PISP being instructed on a specific URL where this supposed user should be redirected. This URL should be a place where the user can prove their identity (e.g., by logging in).



1.4. Authentication

secret will then be passed along to the PISP so that the PISP can demonstrate a chain of trust:

- The DFSP trusts the User
- The DFSP gives the User a secret
- The User trusts the PISP
- The User gives the PISP the secret that came from the DFSP
- The PISP gives the secret to the DFSP
- The DFSP verified that the secret is correct

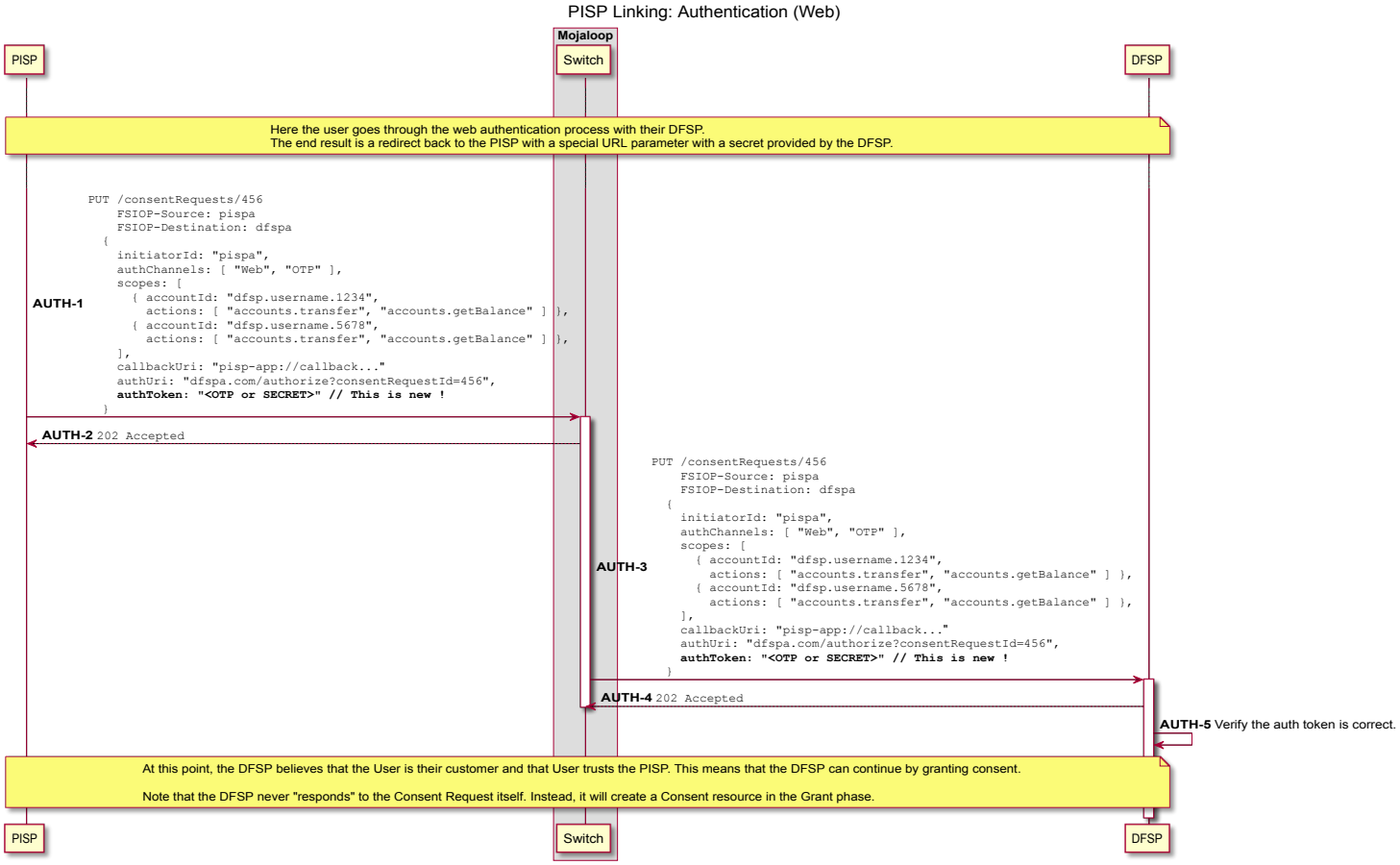
This chain results in the conclusion: The DFSP can trust the PISP is acting on behalf of the User, and mutual trust exists between all three parties.

The process of establishing this chain of trust depends on the authentication channel being used:

1.4.1. Web

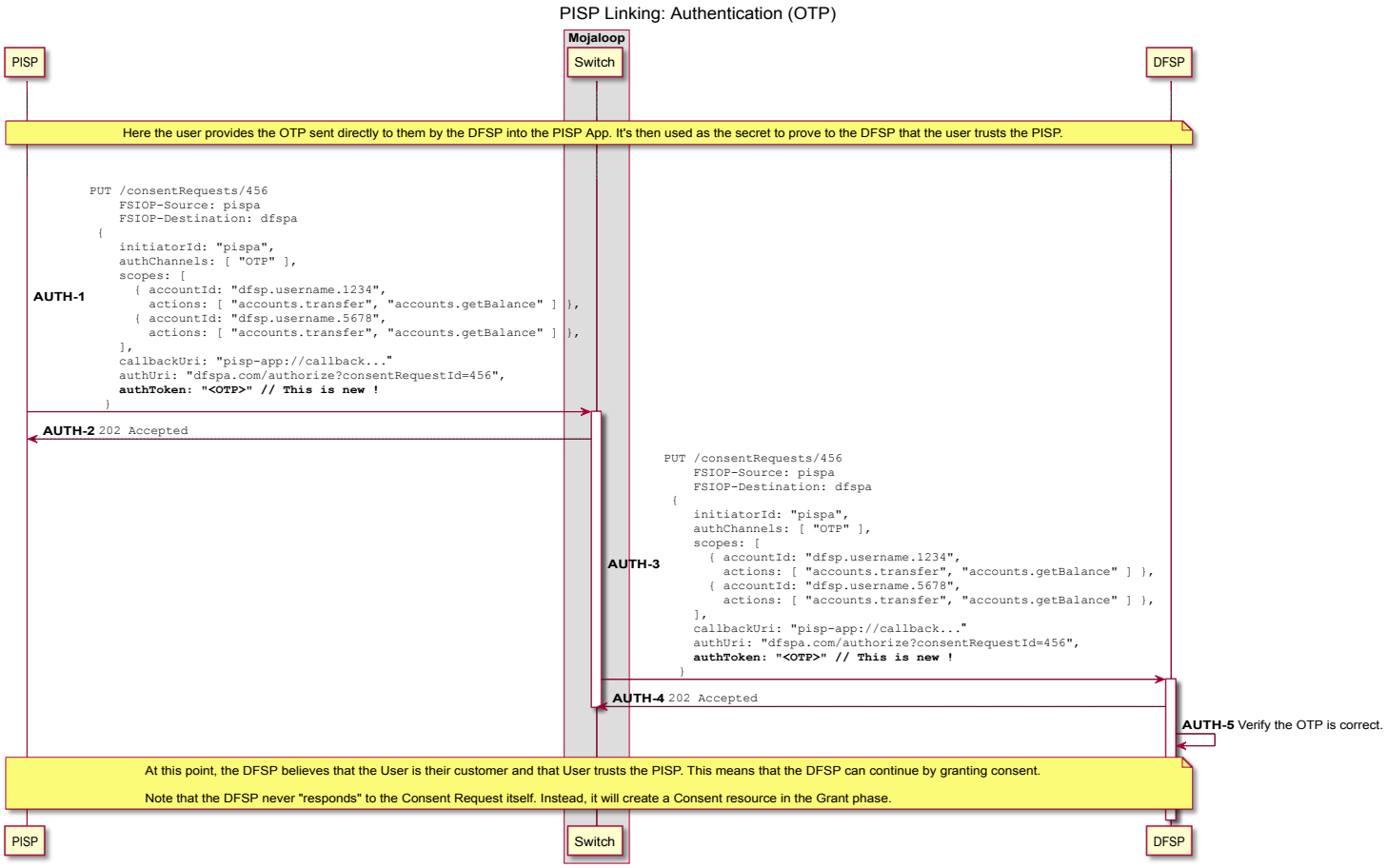
In the web authentication channel, the user is actually redirected to the DFSP's website where they can prove their identity (likely by a typical username and password style login).

Note: Keep in mind that at this stage, the User may update their choices of which accounts to link with. The result of this will be seen later on when during the Grant consent phase, where the DFSP will provide the correct values to the PISP in the `scopes` field.



1.4.2. OTP

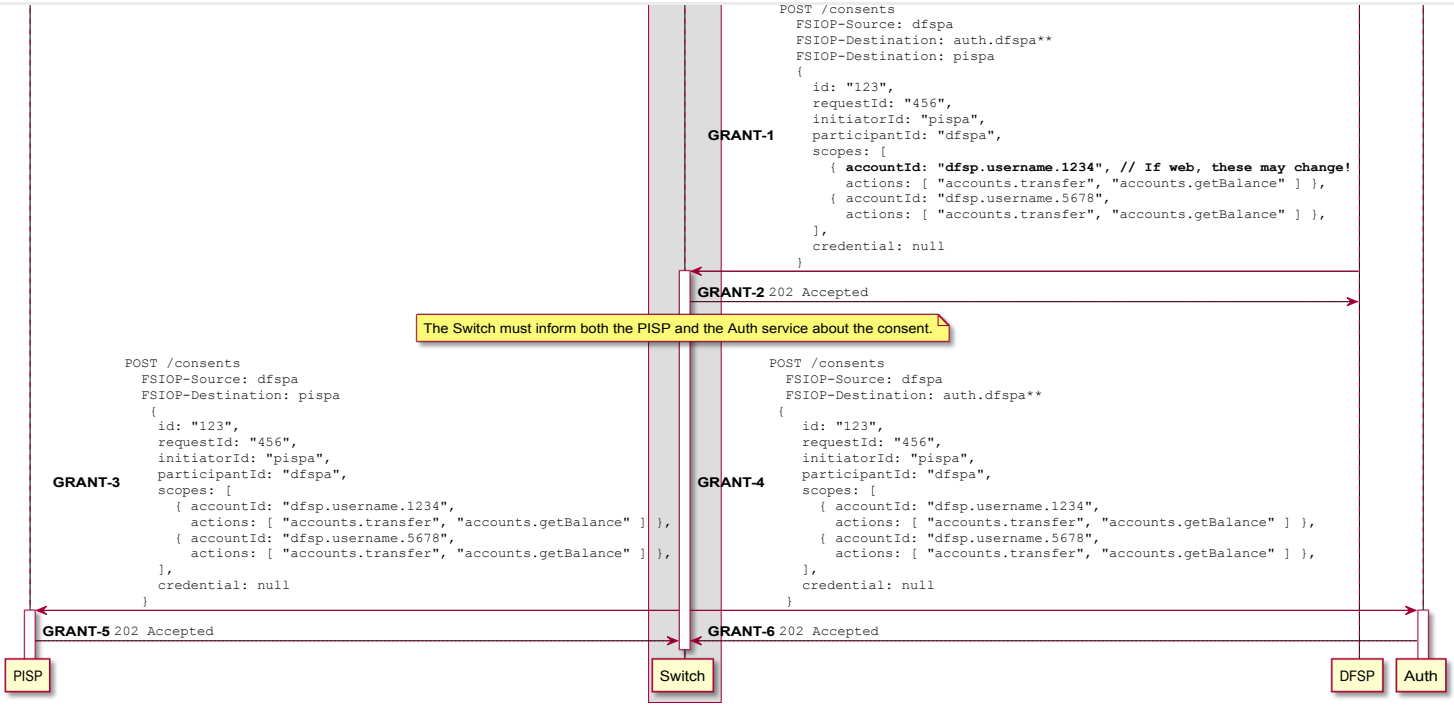
When using the OTP authentication channel, the DFSP will send the User some sort of one-time password over a pre-established channel (most likely SMS). The PISP should prompt the user for this secret and then provide that back to the DFSP.



1.5. Grant consent

Now that mutual trust has been established between all three parties, the DFSP is able to create a record of that fact by creating a new Consent resource. This resource will store all the relevant information about the relationship between the three parties, and will eventually contain additional information for how the User can prove that it consents to each individual transfer in the future.

This phase consists exclusively of the DFSP requesting that a new consent be created. This request must be conveyed both to the PISP itself and the Auth service which will be the record of trust for these resources.



1.6. Credential registration

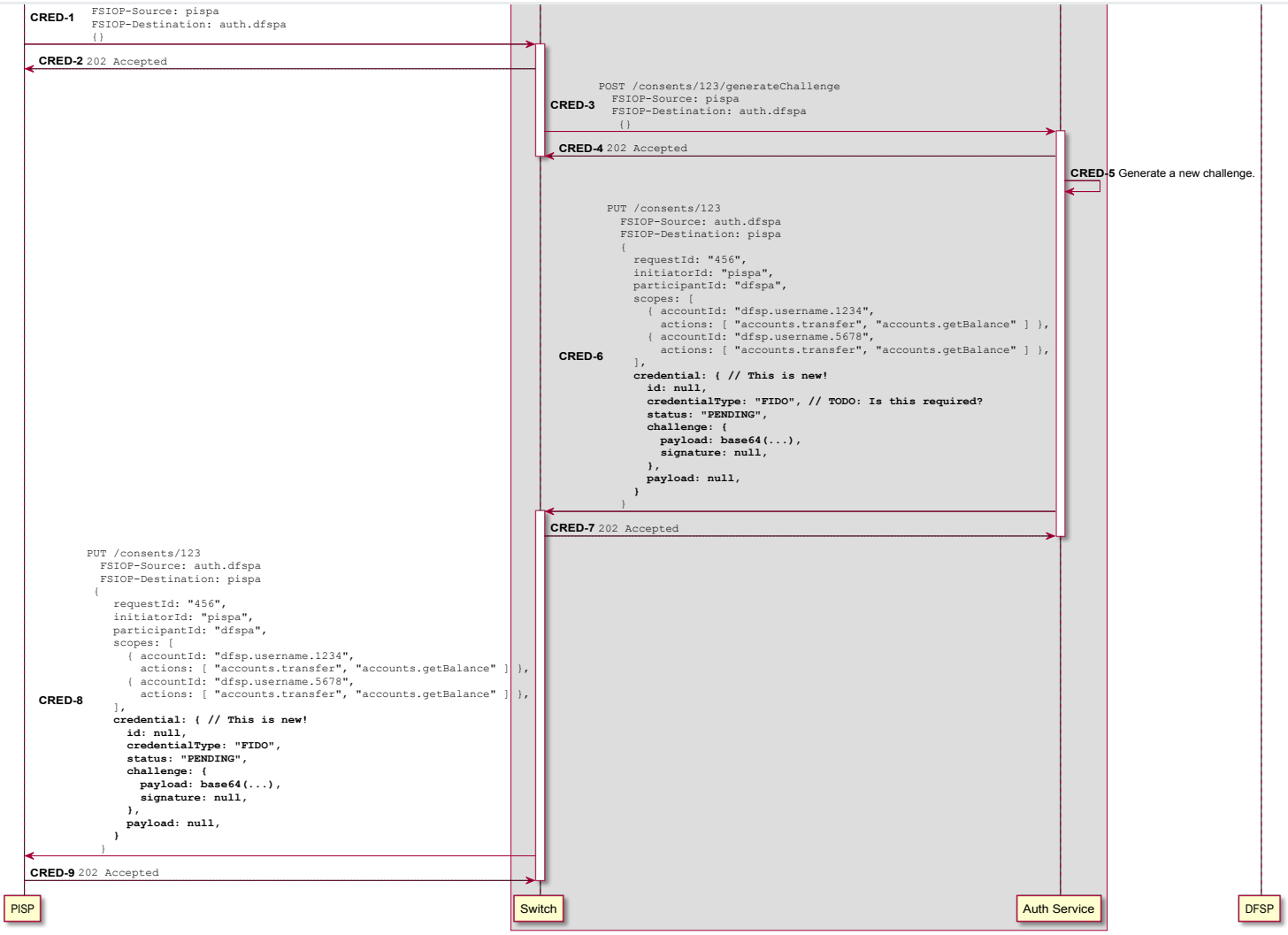
Once the consent resource has been created, the PISP will attempt to establish with the network the credential that should be used to verify that a user has given consent for each transfer in the future.

This will be done by storing a FIDO credential (e.g., a public key) on the Auth service inside the consent resource. When future transfers are proposed, we will require that those transfers be digitally signed by the FIDO credential (in this case, the private key) in order to be considered valid.

This credential registration is composed of two phases: requesting a challenge and finalizing the signature.

1.6.1. Requesting a challenge

In this sub-phase, the PISP requests a challenge from the Auth service, which will be returned to the PISP via a `PUT /consents/{ID}` API call.

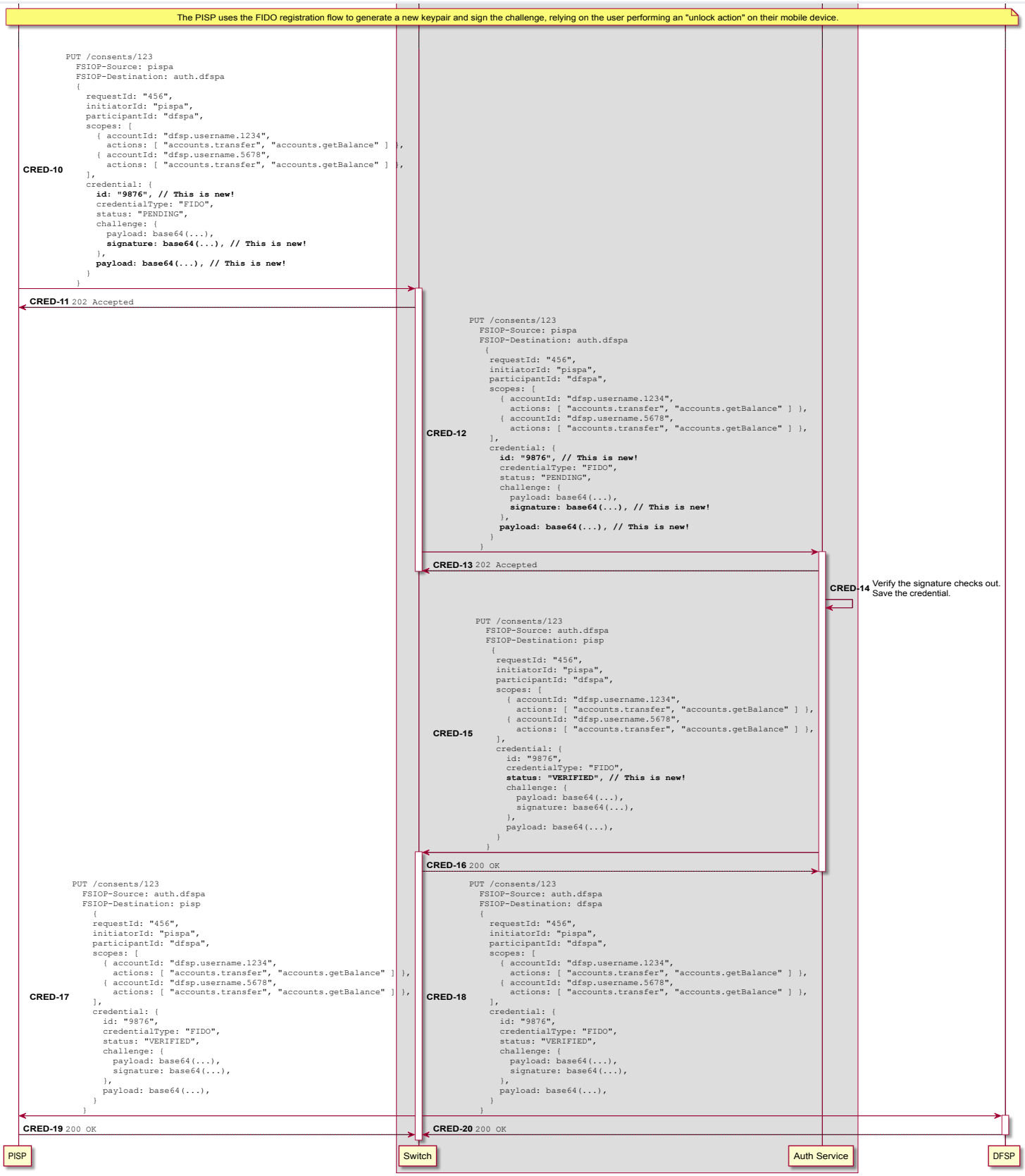


1.6.2. Finalizing the credential

Once the challenge is provided to the PISP, the PISP will generate a new credential on the device, digitally sign the challenge, and provide the some new information about the credential on the Consent resource:

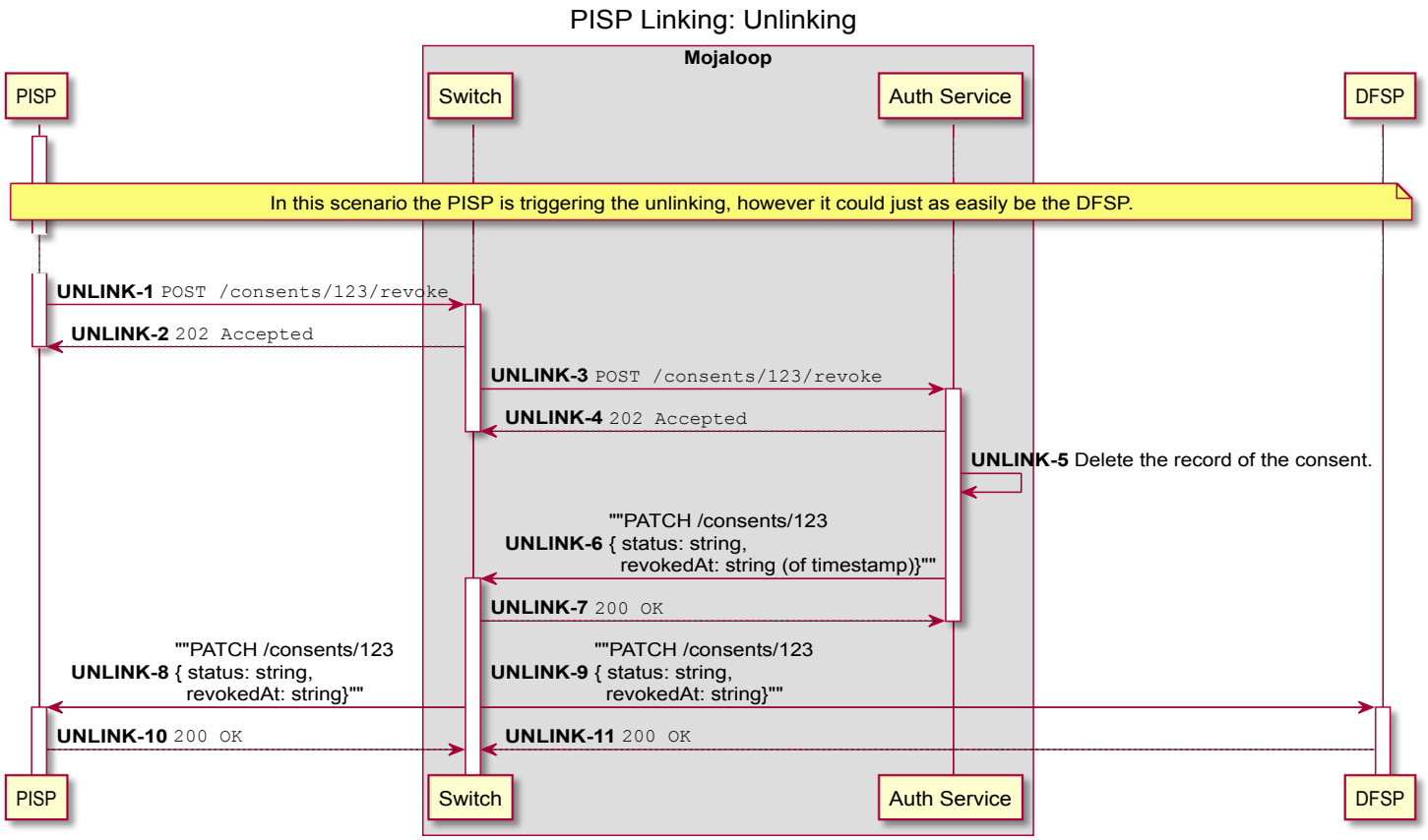
1. The credential itself (the public component)
2. A signature of the challenge (to be verified against this public component)
3. The ID of the credential understood by the device itself

This information is provided back to the Auth service, which then verifies that the signature is correct. It then updates the status of the credential to `"VERIFIED"`, and notifies both the PISP and the DFSP about these new changes to the Consent resource.



2. Unlinking

At some point in the future, it's possible that a User, PISP, or DFSP may decide that the relationship of trust previously established should no longer exist. For example, a very common scenario might be a user losing their mobile device and using an interface from their DFSP to remove the link between the lost device, the PISP, and the DFSP.



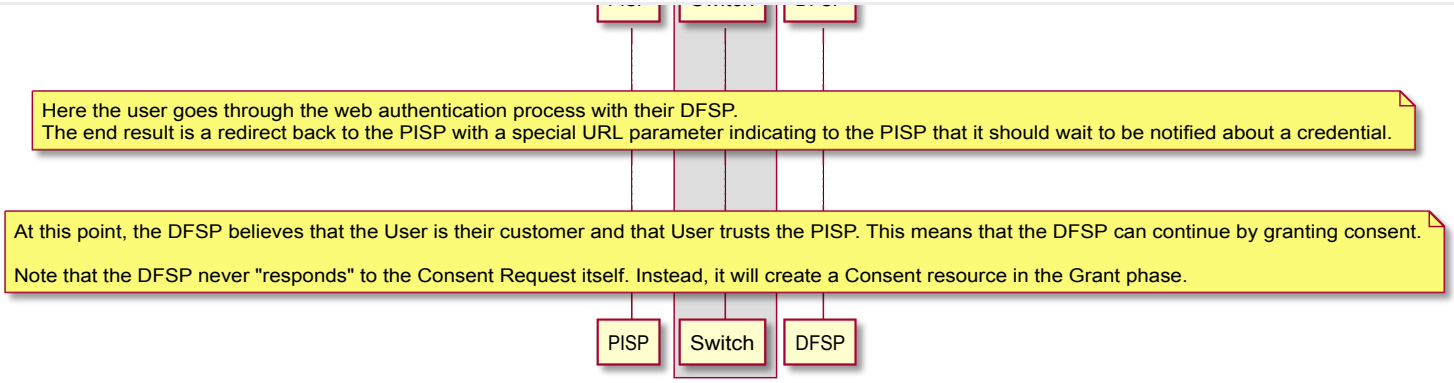
3. Third-party credential registration

There is ongoing work with the FIDO alliance to allow third parties the ability to collect a FIDO credential on behalf of the intended user of the credential. In other words, this work would allow the DFSP (during the web authentication flow) to ask the user to provide a FIDO credential that would be for use by the PISP rather than the DFSP.

If this becomes possible, the flow changes, speciifcally in the Authentication phase and the Credential registration phase.

3.1. Authentication

The authentication phase becomes very minimal. Since the credential will be collected by the DFSP itself (for use later by the PISP), there's no need to send back any sort of secret and no need to pass a secret back to the DFSP.



3.2. Credential registration

TODO!

Edit this page on GitHub

Participant Endpoint Enums

This is the list of endpoints that a PISP or DFSP must register with the switch to get callbacks from the `thirdparty-api-adapter` and `transaction-request-service`.

These endpoints should be added to the enums of `central-services-shared`, and the seeds of `central-ledger`

PISP:

- `THIRDPARTY_CALLBACK_URL_CONSENT_REQUEST_PUT`
- `THIRDPARTY_CALLBACK_URL_CONSENT_REQUEST_PUT_ERROR`
- `THIRDPARTY_CALLBACK_URL_CONSENT_POST`
- `THIRDPARTY_CALLBACK_URL_CONSENT_PUT`
- `THIRDPARTY_CALLBACK_URL_CONSENT_PUT_ERROR`
- `THIRDPARTY_CALLBACK_URL_TRANSACTION_REQUEST_PUT`
- `THIRDPARTY_CALLBACK_URL_TRANSACTION_REQUEST_PUT_ERROR`
- `FSPIOP_CALLBACK_URL_TRX_REQ_SERVICE`

> (Note this is the existing endpoint for `/authorizations` resource)

DFSP:

- `THIRDPARTY_CALLBACK_URL_TRANSACTION_REQUEST_POST`
- `THIRDPARTY_CALLBACK_URL_CONSENT_REQUEST_PUT`
- `THIRDPARTY_CALLBACK_URL_CONSENT_REQUEST_PUT_ERROR`
- `THIRDPARTY_CALLBACK_URL_CONSENT_POST`

> Point to central-auth to get the final consent creation - see GRANT-4

- `THIRDPARTY_CALLBACK_URL_CREATE_CREDENTIAL_POST`

> Point to central-auth for central, or dfsp for self-hosted auth service

- `THIRDPARTY_CALLBACK_URL_CONSENT_PUT`

- `THIRDPARTY_CALLBACK_URL_CONSENT_PUT_ERROR`
- `FSPIOP_CALLBACK_URL_TRX_REQ_SERVICE`

  > (Note this is the existing endpoint for `/authorizations` resource)

- `THIRDPARTY_CALLBACK_URL_TRANSACTION_REQUEST_AUTHORIZATIONS_POST`

  > Point to central-auth for central, or dfsp for self-hosted auth service

- `THIRDPARTY_CALLBACK_URL_TRANSACTION_REQUEST_AUTHORIZATIONS_PUT`

  > Point to dfsp

- `THIRDPARTY_CALLBACK_URL_TRANSACTION_REQUEST_AUTHORIZATIONS_PUT_ERROR`

  > Point to dfsp

Scheme Adapter Changes

A design document to explain/discuss the changes required in the `sdk-scheme-adapter` to implement PISP functionality.

Scheme Adapter Background

One of the purposes of the `sdk-scheme-adapter` is to abstract away the Mojaloop API's async nature to make integrations easier. In addition, we use the scheme adapter as part of the `mojaloop-simulator`, so in order to prove and test our solution, we need to add functionality to the `mojaloop-simulator`, and hence the `sdk-scheme-adapter`.

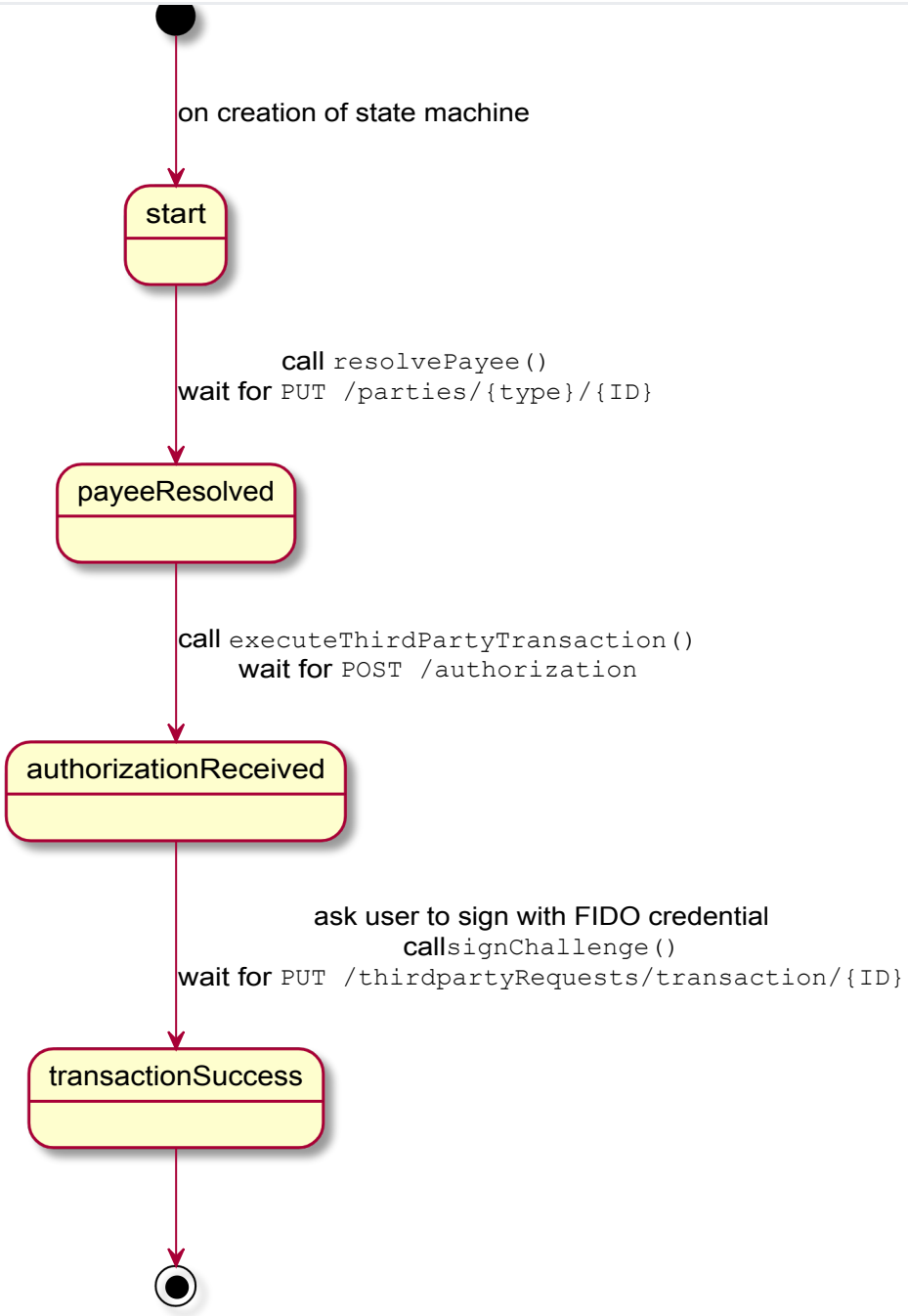
State Machines

The `sdk-scheme-adapter` uses a series of state machines, implemented using the javascript-state-machine library. These state machines make it possible to handle complex async calls to a mojaloop hub, and expose them as a simpler, syncronous API to DFSPs and the `mojaloop-simulator` alike.

Below we propose a number of new state machines to be implemented as a part of the PISP functionality. Keep in mind the names are not set in stone, this is very much a first pass and will be expected to change.

OutboundThirdpartyTransactionModel

Purpose:
Models the PISP side of a PISP transaction, starting with a `GET /parties`, all the way to the `PUT /thirdpartyRequests/transaction/{ID}` callback.



States:

- `start` - when the state machine is created
- `payeeResolved` - on a `PUT /parties/{type}/{ID}`
- `authorizationReceived` - on a `POST /authorization`
- `transactionSuccess` - on a `PUT /thirdpartyRequests/transaction/{ID}`
- `error` - on any error callback, or internal processing error

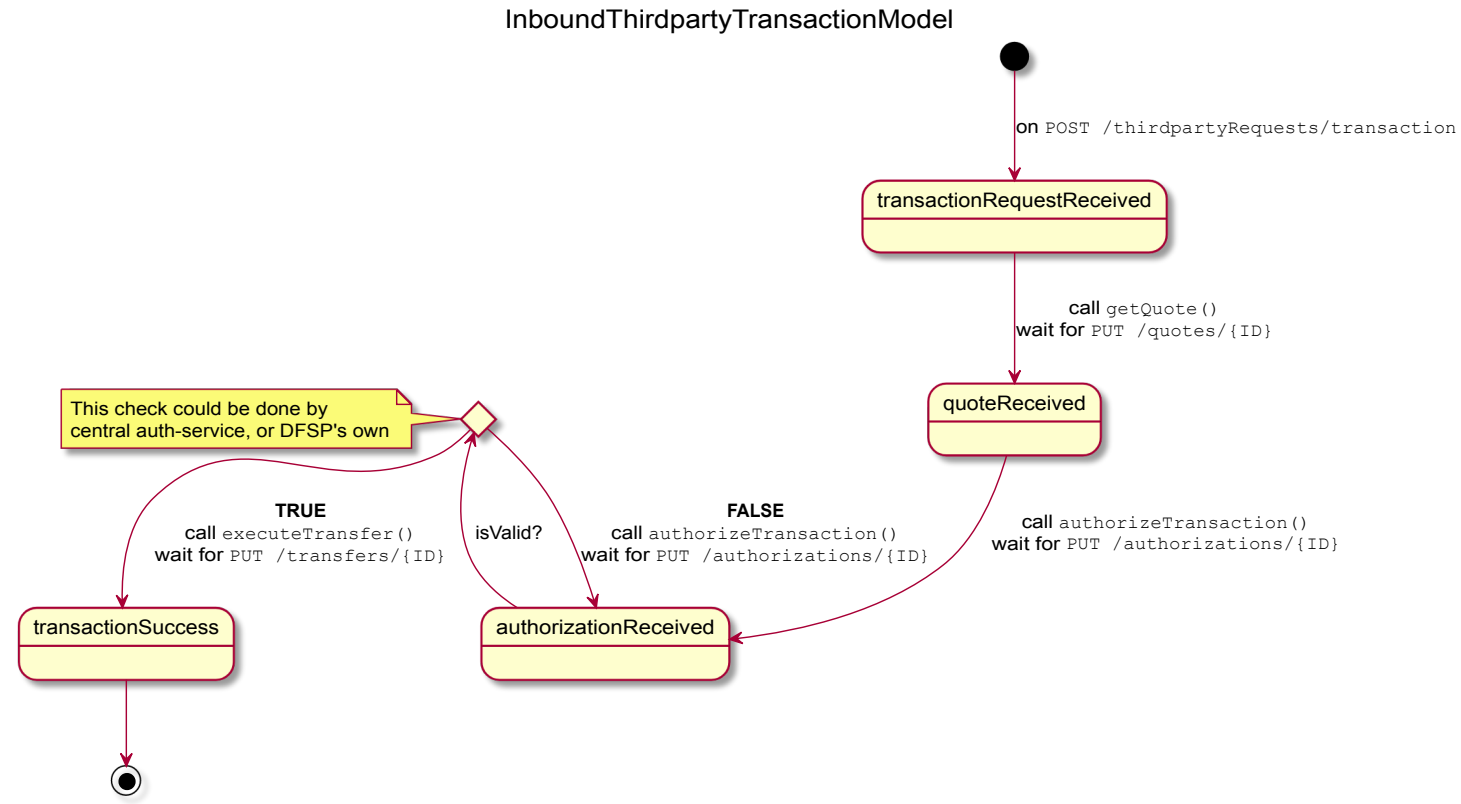
Functions:

- `resolvePayee()` - Calls `GET /parties/{type}/{ID}` to lookup the payee party of the transaction
- `executeThirdPartyTransaction()` - Calls `POST /thirdpartyRequests/transaction` to kick off the PISP Transaction

InboundThirdpartyTransactionModel

Purpose: Models the DFSP side of a PISP transaction, initiated by receiving a `POST /thirdpartyRequests/transaction`

Model:



States:

- `transactionRequestReceived` - on a `POST /thirdpartyRequests/transaction`
- `quoteReceived` - on a `PUT /quotes/{ID}`
- `authorizationReceived` - on a `PUT /authorizations/{ID}` , with a signed challenge
- `transactionSuccess` - on a `PUT /transfers/{ID}`
- `error` - on any error callback, or internal processing error

Functions:

- `getQuote()` - Calls `POST /quotes` to ask the payee for a quote for the given transaction
- `authorizeTransaction()` - Calls `POST /authorizations` to ask the PISP to ask their user to authorize the transaction with their FIDO credential
- `executeTransfer()` - Calls `POST /transfer` to execute the transfer

Questions:

- How do we handle authorization retries in the above models? Maybe we can leave this for now.

Mojaloop/PISP Roles

A summary of Mojaloop FSPIOP API Endpoints and Roles as they stand today.

Roles

- DFSP
  - fund-holding participant
  - has clearing and settlement functions
  - allows PISPs to initiate transfers on behalf of their users
- PISP
  - non-fund holding participant (and therefore no clearing or settlements as well)
  - initiates transfers on user's behalf
  - assumes delegated permissions *from* user

API Calls:

- Existing calls are taken from Mojaloop API Spec
- New calls are detailed either in the updated pisp flows, or #59-mojaloop-specification

API Calls - Outbound (From Participant -> Switch)

FSPIOP-API (DFSP -> Switch)

Name	VERB	Resource	DFSP	PISP	Note
Create Bulk Participant Information	`PUT`	`/participants/{ID}`	✅	❌	
Lookup Participant Information	`GET`	`/participants/{Type}/{ID}*`	✅	❌	
Create Participant Information	`POST`	`/participants/{Type}/{ID}*`	✅	❌	

Delete Participant Information	DELETE	/participants/{Type}/{ID}*	✅	❌	
Lookup Party Information	GET	/parties/{Type}/{ID}*	✅	✅	
Retrieve Transaction Request Information	GET	/transactionRequests/{ID}	✅	❌	
Perform Transaction Request	POST	/transactionRequests	✅	❌	
Retrieve Quote Information	GET	/quotes/{ID}	✅	❌	
Calculate Quote	POST	/quotes	✅	❌	
Retrieve Bulk Quote Information	GET	/bulkQuotes/{ID}	✅	❌	
Calculate Bulk Quote	POST	/bulkQuotes	✅	❌	
Perform Authorization	GET	/authorizations/{ID}	✅	❌	
Update Authorization	PUT	/authorizations/{ID}	✅	✅	
Retrieve Transfer Information	GET	/transfers/{ID}	✅	❌	
Perform Transfer	POST	/transfers	✅	❌	
Retrieve Bulk Transfer Information	GET	/bulkTransfers/{ID}	✅	❌	
Perform Bulk Transfer	POST	/bulkTransfers	✅	❌	
Retrieve Transaction Information	GET	/transactions/{ID}	✅	❌	

> * also: `{Type}/{ID}/{SubId}`

ThirdParty-PISP-API (PISP -> Switch)

Name	isNew?	VERB	Resource	DFSP
Lookup Party Information	N	GET	/parties/{Type}/{ID}*	✅
Update Authorization	N	PUT	/authorizations/{ID}	✅
Create Consent Request	Y	POST	/consentRequests	❌
Update Consent Request	Y	PUT	/consentRequests/{ID}	✅
Request Challenge	Y	PUT	/consents/{ID}/createCredential	❌
Lookup Consent	Y	GET	/consents/{ID}	✅
Update Consent	Y	PUT	/consents/{ID}	✅
Initiate 3rd party request	Y	POST	/thirdpartyRequests/transactions	❌
Get 3rd party request information	Y	GET	/thirdpartyRequests/transactions/{ID}	❌

> * also: {Type}/{ID}/{SubId}

ThirdParty-DFSP-API (DFSP -> Switch)

Api for a DFSP to implement to allow PISP functionality.

Name	isNew?	VERB	Resource

Party Authorization	Y	POST	/authorizations
Update Consent Request	Y	PUT	/consentRequests/{ID}
Create Consent	Y	POST	/consents
Lookup Consent	Y	GET	/consents/{ID}
Update Consent	Y	PUT	/consents/{ID} *
Verify a 3rd party transaction	Y	POST	/thirdpartyRequests/transactions/{ID/authorizatio

> * `PUT /consents/{ID}` from the DFSP may not be required.

API Calls - Inbound (From Switch -> Participant)

FSPIOP-API (Switch -> DFSP)

Name	VERB	Resource	DFSP	PISP	Note
Return Bulk Participant Information	PUT	/participants/{ID}	✅	❌	
Return Bulk Participant Information Error	PUT	/participants/{ID}/error	✅	❌	
Return Participant Information	PUT	/participants/{Type}/{ID}*	✅	❌	

Return Participant Information Error	PUT	/participants/{Type}/{ID}/error*	✅	❌	
Return Party Information	PUT	/parties/{Type}/{ID}*	✅	✅	PISP needs to get result of party lookup
Return Party Information Error	PUT	/parties/{Type}/{ID}/error*	✅	✅	PISP needs to get result of party lookup
Lookup Party Information	GET	/parties/{Type}/{ID}*	✅	✅	
Return Transaction Request Information	PUT	/transactionRequests/{ID}	✅	❌	
Return Transaction Request Information Error	PUT	/transactionRequests/{ID}/error	✅	❌	
Return Quote Information	PUT	/quotes/{ID}	✅	❌	
Return Quote Information Error	PUT	/quotes/{ID}/error	✅	❌	
Return Bulk Quote Information	PUT	/bulkQuotes/{ID}	✅	❌	
Return Bulk Quote Information Error	PUT	/bulkQuotes/{ID}/error	✅	❌	
Return Authorization Result	PUT	/authorizations/{ID}	✅	✅	
Return Authorization Error	PUT	/authorizations/{ID}/error	✅	✅	

Return Transfer Information	PUT	/transfers/{ID}	✅	❌	
Return Transfer Information Error	PUT	/transfers/{ID}/error	✅	❌	
Return Bulk Transfer Information	PUT	/bulkTransfers/{ID}	✅	❌	
Return Bulk Transfer Information Error	PUT	/bulkTransfers/{ID}/error	✅	❌	
Return Transaction Information	PUT	/transactions/{ID}	✅	❌	
Return Transaction Information Error	PUT	/transactions/{ID}/error	✅	❌	

ThirdParty-PISP-API (Switch -> PISP)

Name	isNew?	VERB	Resource	D
Return Party Information	N	PUT	/parties/{Type}/{ID}*	✅
Return Party Information Error	N	PUT	/parties/{Type}/{ID}/error*	✅
Return Authorization Result	N	PUT	/authorizations/{ID}	✅
Return Authorization Error	N	PUT	/authorizations/{ID}/error	✅
Update Consent Request	Y	PUT	/consentRequests/{ID}	✅
Update Consent Request Error	Y	PUT	/consentRequests/{ID}/error	✅

Create Consent	Y	POST	/consents/ *2	
Create Or Update Consent	Y	PUT	/consents/{ID} *2	
Perform Authorization	Y	POST	/authorizations/	
Update 3rd Party Transaction Request	Y	PUT	/thirdpartyRequests/transactions/{ID}	
Update 3rd Party Transaction Request Error	Y	PUT	/thirdpartyRequests/transactions/{ID}/error	

> * also: {Type}/{ID}/{SubId} *2 We may want this to be *only* be a POST to create a consent, to keep things consistent

ThirdParty-DFSP-API (Switch -> DFSP)

DFSP inbound API calls required for PISP functionality.

Name	isNew?	VERB	Resource
Lookup Party Information	N	GET	/parties/{Type}/{ID}*
Return Quote Information	N	PUT	/quotes/{ID}
Perform Transfer	N	POST	/transfers
Update Authorization	N	PUT	/authorizations/{ID}
Create Consent Request	Y	POST	/consentRequests

Update Consent Request	Y	`PUT`	`/consentRequests/{ID}`
Lookup Consent	Y	`GET`	`/consents/{ID}`
Update Consent	Y	`PUT`	`/consents/{ID}` *2
Initiate 3rd party request	Y	`POST`	`/thirdpartyRequests/transactions`
Update a 3rd party verification	Y	`PUT`	`/thirdpartyRequests/transactions/{ID/authorizatio`
Update a 3rd party verification error	Y	`PUT`	`/thirdpartyRequests/transactions/{ID/authorizatio`

> * also: `{Type}/{ID}/{SubId}` *2 `PUT /consents/{ID}` from the DFSP may not be required.

Security Considerations:

We assume an API Gateway will be able to distinguish between the participant's role (DFSP or PISP), and whether or not they have the access to call the given API.

`PATCH /thirdpartyRequests/transactions/{id}` Callback Design

1. Subscription

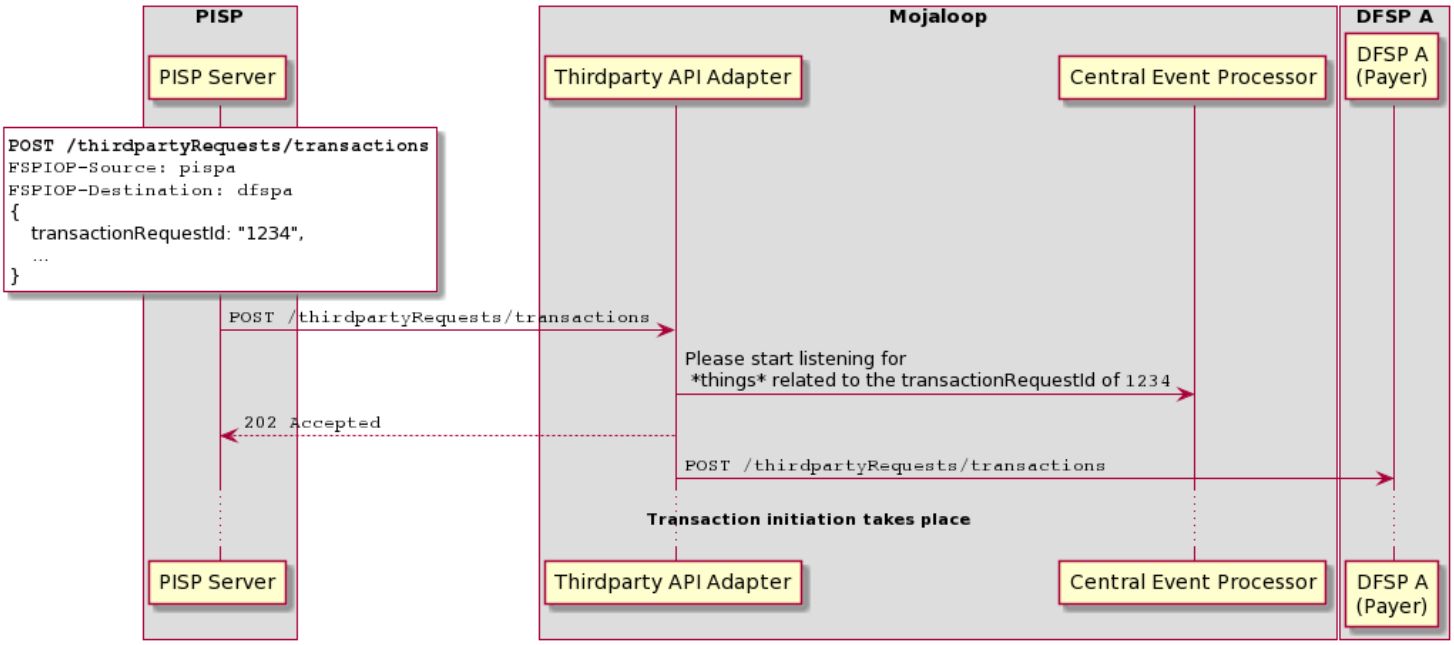
The goal of subscription is to allow the Central-Event-Processor (CEP) to listen for **ThirdpartyAuthorizationRequest** Events on the Notifications Topic, and deliver them to the Thirdparty-API-Adapter.

1.1 `POST /thirdpartyRequests/transactions`

The PISP issues a `POST /thirdpartyRequests/transactions` request to dfspA, asking to transfer funds from their users' account to dfspB.

The Thirdparty-API-Adapter recieves this request, and emits a **ThirdpartyTransactionRequest Subscription** event

The CEP recieves this event, and starts listening for other events related to `transferRequestId=1234` on the Notifications Topic.



2. Context Gathering

successful 3rd party initiation, it needs to gather more context.

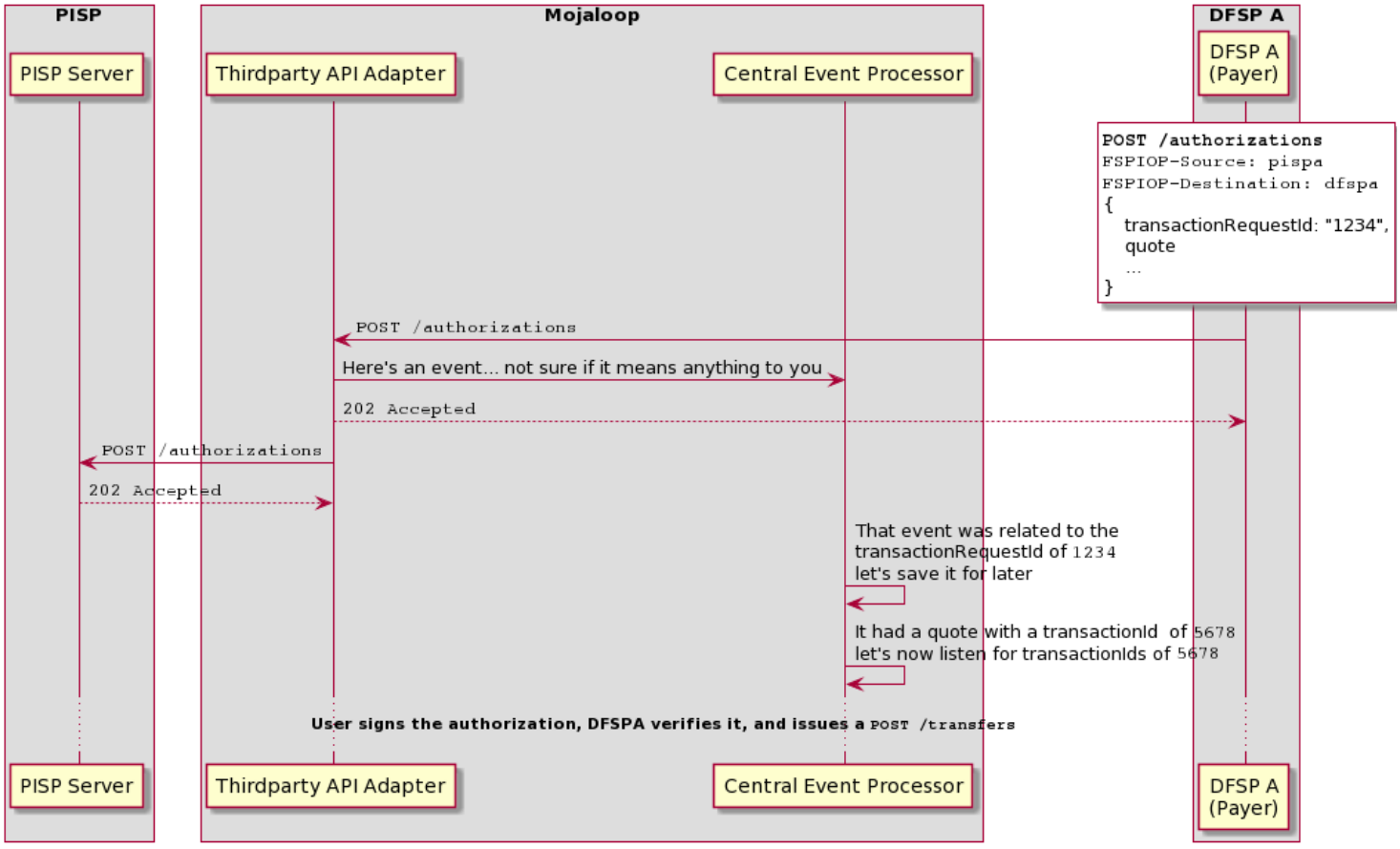
2.1 `POST /authorizations`

> Maybe this should be on the quote you say? See outstanding-questions

The Thirdparty-API-Adapter recieves a `POST /authorizations` request, and before forwarding it to the PISP, it emits a **ThirdpartyAuthorizationRequest** Event to the notifications kafka topic.

The since the CEP is listening for events related to `transactionRequestId=1234`, is observes that this event is related.

It inspects the event body, and sees the `body.transactionId` of `5678`. CEP starts listening for events related to the `transactionId=5678`.

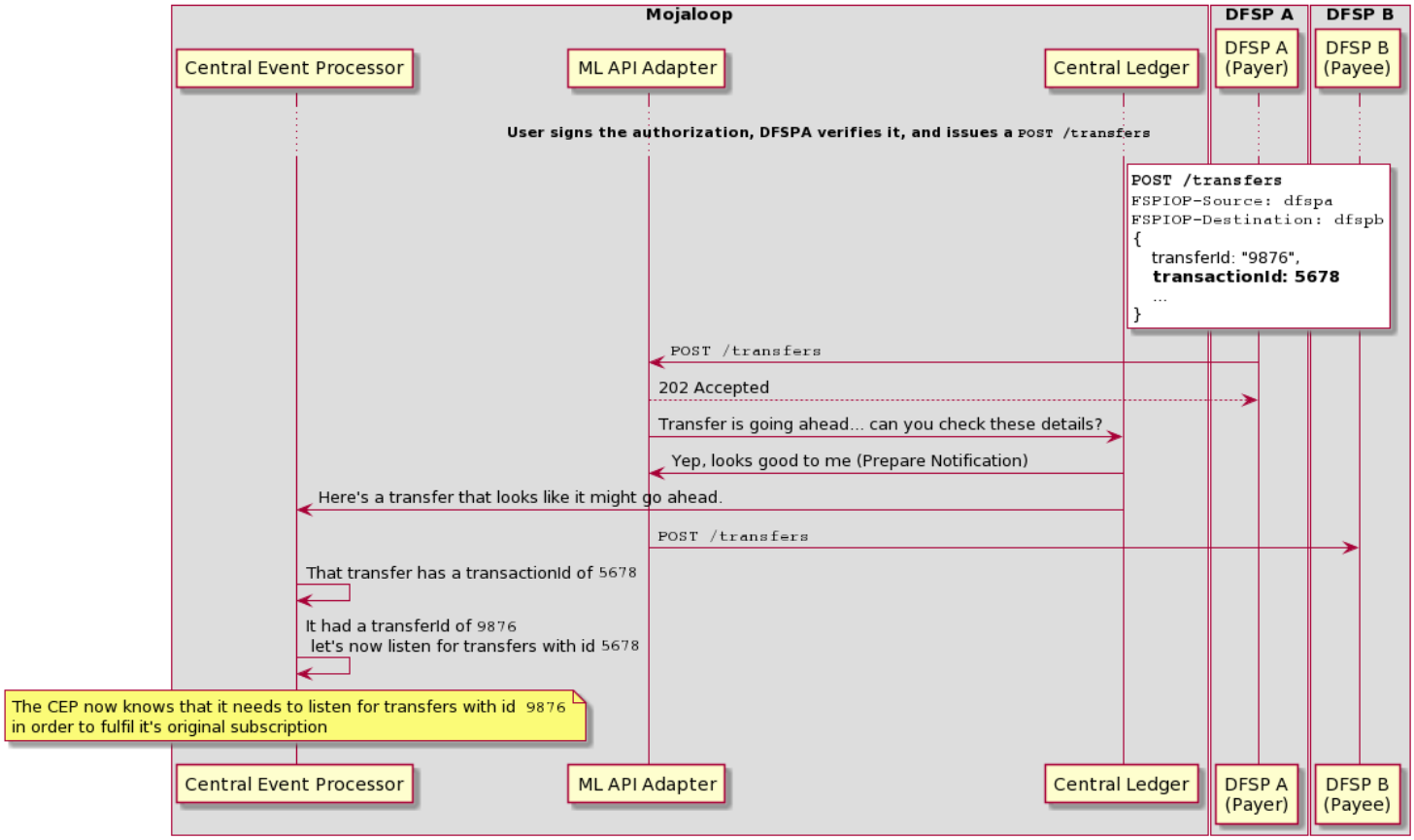


2.2 `POST /transfers`

DFSPA issues a `POST /transfer` request to initiate the transfer.

The central-ledger processes this call, and emits a **Transfer Prepare Notification** event

The CEP observes this event (as it contains either a request body or encoded interledger packet with a `transactionId=5678`)



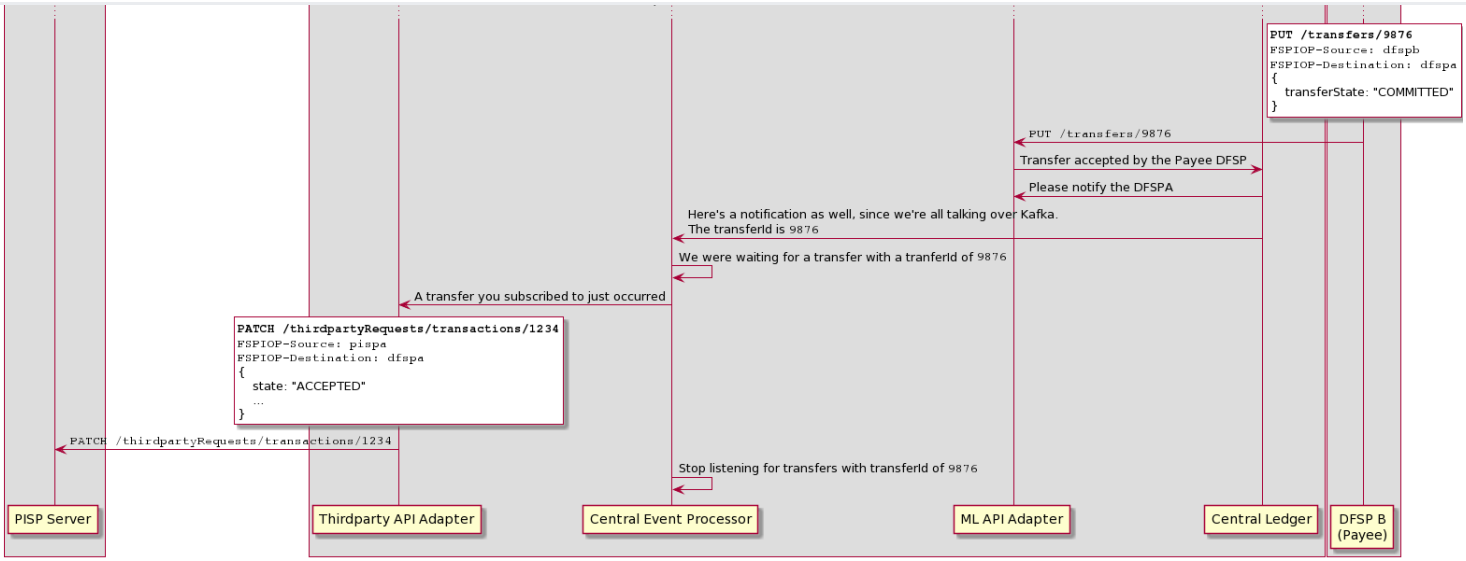
3. Notification

DFSPB issues a `PUT /transfers/{id}` .

The transfer is processed by the central-ledger, which emits a **Transfer Fulfil Notification** event.

The CEP sees this **Transfer Fulfil Notification** event, with a `transferId=9876` . It sees the related `transactionRequest` listener, and emits a **ThirdpartyTransactionRequest Fulfil** Event.

The Thirdparty-API-Adapter sees this event, and sends a `PATCH /thirdpartyRequests/transactions/1234` request to the PISP.



Outstanding Questions

1. Should the CEP be listening for `POST /quotes` (probably emitted by the quoting-service) or for `POST /authorizations` (probably emitted by the 3p-api-adapter?)

- `POST /authorizations` is more efficent, since it certainly contains a `transactionRequestId`, whereas a quote *may* contain a `transactionRequestId`
- `POST /quotes` is more generic, and could lead us to better design decisions (such as not including the quote response *in* the `POST /authorizations` body) in the future
  - does the `quoting-service` publish to kafka at the moment? No.

2. Can we generalize this pattern better and make it more applicable to other use cases?

- e.g. `PISP Consents`, `FX`, `Cross-network`?
  - Probably, we should go through this design process with those use cases and see the commonalities. That can be covered in a separate document.

3. When should does the CEP *stop listening*?

- We need to enumerate the error conditions a little better

4. When do we consider a *transaction* final? Is it determined by the PayeeFSP? Or perhaps by the Central-Ledger? Or could it be either?

- I *think* it's based on the Central-Ledger's **Transfer Fulfil Notification**

5. How tightly or loosely coupled should the different listeners be?

- Should the listener for `transferId=9876` *know* about the listener for `transferRequestId=1234`?

6. Should this all be on the Notifications topic? Or should there be some division of topics?

- I don't want to burden the ml-api-adapter's notifications topic handler with extra notifications
- We can make use of existing different group ids...

7. What database should the CEP use to store these subscriptions?

- It currently uses Mongo, but that may be unsuitable
- Multiple listeners referring to 1 shared object may also be tricky, but likely necessary... so we may want some transaction guarantees

Appendix A - List of Kafka Events

A.1 ThirdpartyTransactionRequest Subscription

```js
{
  id: '<message.id>',
  from: '<message.initiatorId>',
  to: '<message.payerFsp>',
  type: 'application/json',
  content: {
    headers: '<message.headers>',
    payload: {
      transactionRequestId: '<uuid>',
      sourceAccountId: 'string',
      consentId: 'string',
      ...
    }
  },
  metadata: {
    event: {
      id: '<uuid>',
      type: 'transactionRequest',
      action: 'subscription',
      createdAt: '<timestamp>',
      state: {
        status: 'success',
        code: 0
      }
    }
  }
}
```

A.2 ThirdpartyAuthorizationRequest

```js
{
  id: '<message.id>',
  from: '<message.payerFsp>',
  to: '<message.initiatorId>',
  type: 'application/json',
  content: {
    headers: '<message.headers>',
    payload: {
      transactionRequestId: '<uuid>',
      transactionId: '<uuid>',
      amount: {...},
      quote: {...}
    }
  },
  metadata: {
    event: {
      id: '<uuid>',
      type: 'authorizationRequest',
      action: 'subscription',
      createdAt: '<timestamp>',
      state: {
        status: 'success',
        code: 0
      }
    }
  }
}
```

A.3 Transfer Prepare Notification

> Ref 1.3.1-prepare

```js
{
  id: '<transferMessage.transferId>',
  from: '<transferMessage.payerFsp>',
  to: '<transferMessage.payeeFsp>',
  type: 'application/json',
  content: {
    headers: '<transferHeaders>',
    payload: '<transferMessage>'
  },
  metadata: {
```

```
      type: 'transfer',
      action: 'prepare',
      createdAt: '<timestamp>',
      state: {
        status: 'success',
        code: 0
      }
    }
  }
}
```

A.4 Transfer Fulfil Notification

> Ref: 1.3.2-fulfil-position-handler-consume-v1.1

```
{
  id: '<transferMessage.transferId>',
  from: '<transferMessage.payerFsp>',
  to: '<transferMessage.payeeFsp>',
  type: 'application/json',
  content: {
    headers: '<transferHeaders>',
    payload: '<transferMessage>'
  },
  metadata: {
    event: {
      id: '<uuid>',
      responseTo: '<previous.uuid>',
      type: 'transfer',
      action: 'commit' | 'reserve',
      createdAt: '<timestamp>',
      state: {
        status: 'success',
        code: 0
      }
    }
  }
}
```

A.5 ThirdpartyTransactionRequest Fulfil

```
  from: '<???>',
  to: '<message.initiatorId>',
  type: 'application/json',
  content: {
    headers: '<message.headers>',
    payload: {
      transactionRequestId: '<uuid>',
      sourceAccountId: 'string',
      consentId: 'string',
      ...
    }
  },
  metadata: {
    event: {
      id: '<uuid>',
      type: 'transactionRequest',
      action: 'fulfil',
      createdAt: '<timestamp>',
      state: {
        status: 'success',
        code: 0
      }
    }
  }
}
```

Appendix B - CEP ThirdpartyTransactionRequest Subscription Listeners + Object

```
Active Listeners:
- 'transactionRequestId/1234'

Subscription Object:
{
  initiatorId: 'pispA',
  sourceDFSP: 'dfspA',
  transactionRequestId: '1234',
  transactionId: null,
  transferId: null,
}
```

```
Active Listeners:
- 'transactionRequestId/1234'
```

```
{
  initiatorId: 'pispA',
  sourceDFSP: 'dfspA',
  transactionRequestId: '1234',
  transactionId: '5678',
  transferId: null,
}
```

```
Active Listeners:
- 'transactionRequestId/1234'
- 'transferId/9876'

Subscription Object:
{
  initiatorId: 'pispA',
  sourceDFSP: 'dfspA',
  transactionRequestId: '1234',
  transactionId: '5678',
  transferId: '9876'
}
```

Appendix C - CEP State Transitions

Simplified

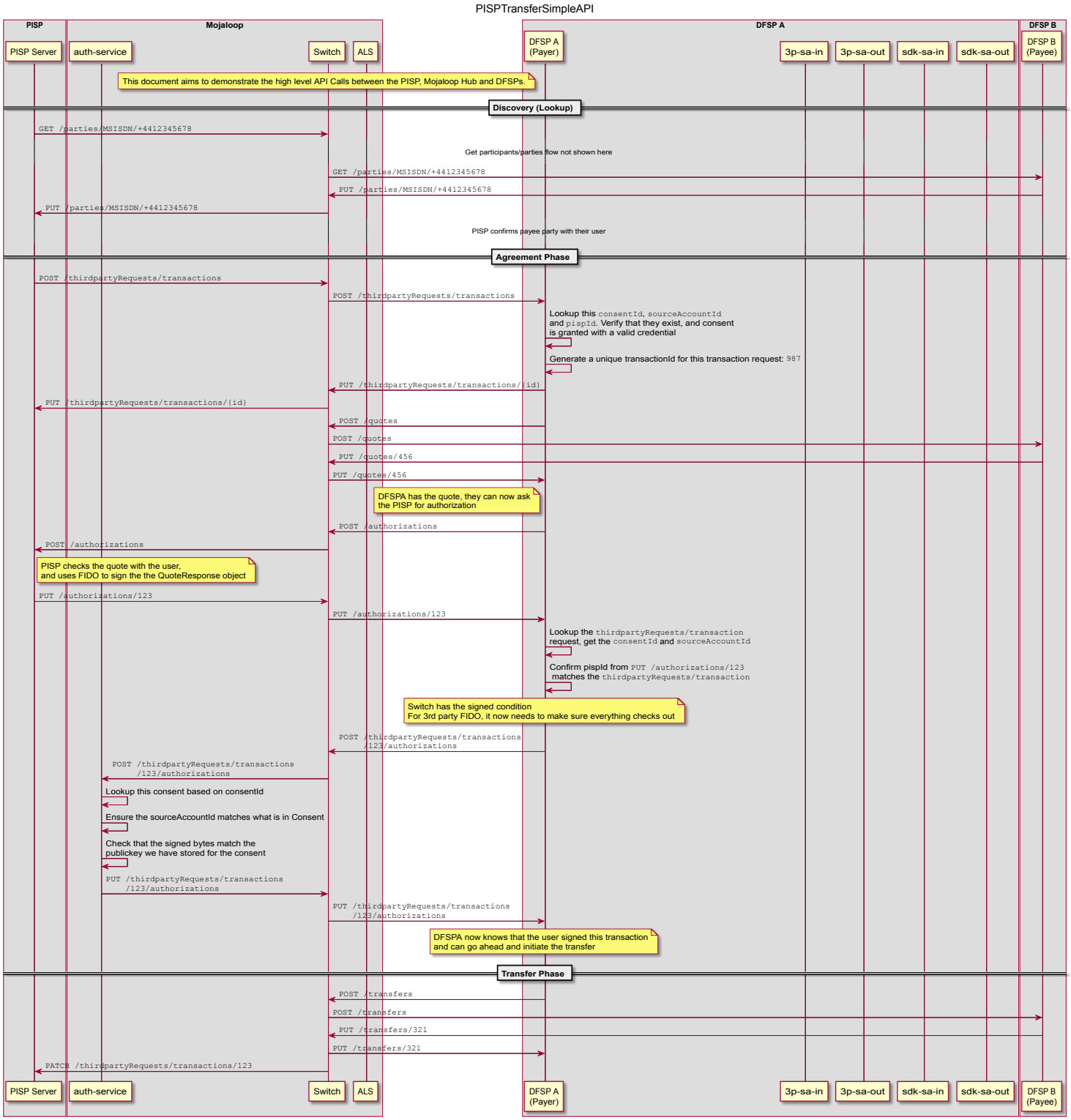
With Errors

Edit this page on GitHub

Transfer API

Happy Path

Edit the transfer flow `.puml` files here: PISP Transfer Api Calls Detailed

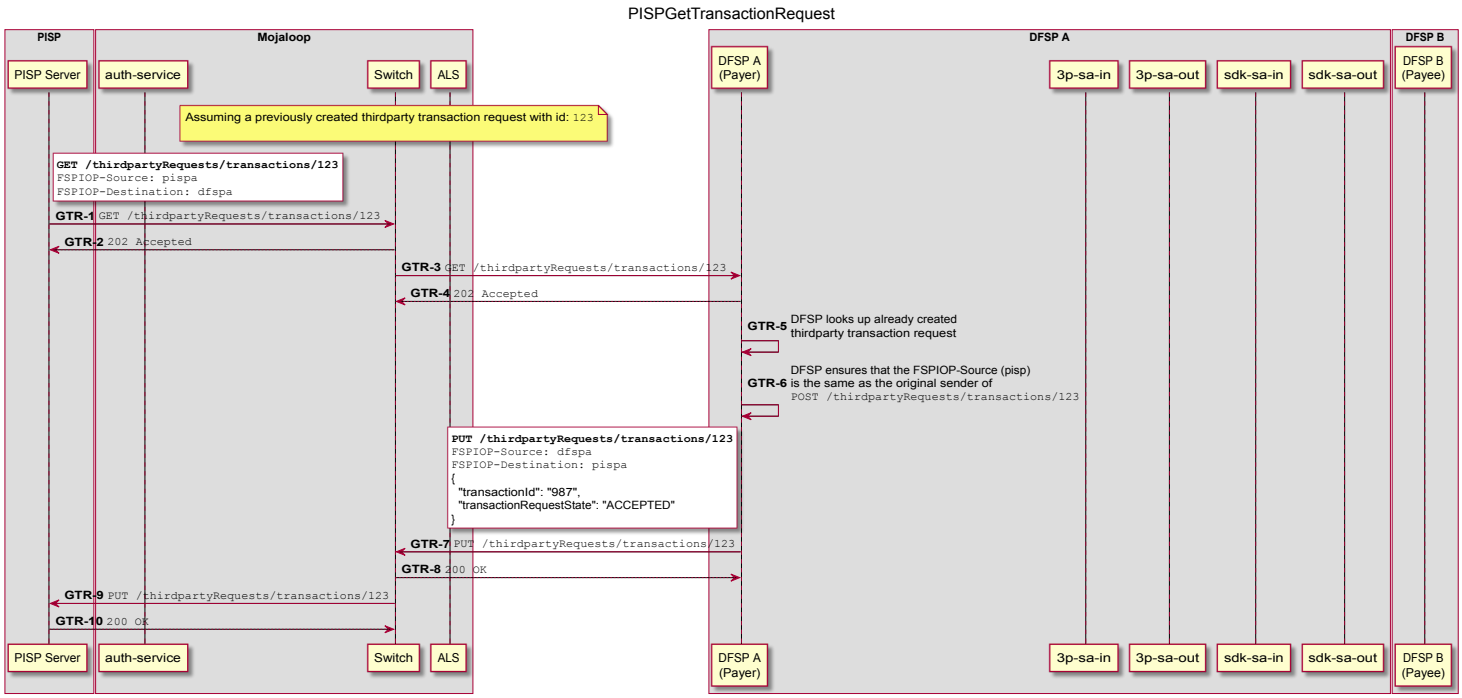


> Puml source: `./transfer/api_calls_simple.puml`

For a more detailed breakdown of the api calls, Refer to the detailed API flows:

Request TransactionRequest Status

A PISP can issue a `GET /thirdpartyRequests/{id}/transactions` to find the status of a transaction request.



1. PISP issues a `GET /thirdpartyRequests/transactions/{id}`
2. Switch validates request and responds with `202 Accepted`
3. Switch looks up the endpoint for `dfspa` for forwards to DFSP A
4. DFSPA validates the request and responds with `202 Accepted`
5. DFSP looks up the transaction request based on it's `transactionRequestId` (`123` in this case)
   - If it can't be found, it calls `PUT /thirdpartyRequests/transactions/{id}/error` to the Switch, with a relevant error message
6. DFSP Ensures that the `FSPIOP-Source` header matches that of the originator of the `POST //thirdpartyRequests/transactions`
   - If it does not match, it calls `PUT /thirdpartyRequests/transactions/{id}/error` to the Switch, with a relevant error message
7. DFSP calls `PUT /thirdpartyRequests/transactions/{id}` with the following request body:

```
  transactionRequestState: TransactionRequestState
}
```

Where `transactionId` is the DFSP-generated id of the transaction, and `TransactionRequestState` is `RECEIVED`, `PENDING`, `ACCEPTED`, `REJECTED`, as defined in 7.5.10 TransactionRequestState of the API Definition

8. Switch validates request and responds with `200 OK`
9. Switch looks up the endpoint for `pispa` for forwards to PISP
10. PISP validates the request and responds with `200 OK`

Error Conditions

The PayerDFSP is responsible for communicating failures to the PISP

1. Thirdparty Transaction Request fails
2. Downstream Quote Failure
3. Authorization Failure
4. Transfer Failure

[todo: in mojaloop/mojaloop#346]