

ETT business architecture

Last updated by | Michaël Piron < PIRONM@Belgrid.net > | Mon, 21 Aug 2023 12:44:03 GMT

1/ Introduction

This page documents the Energy track & Trace business architecture and related API design.

We describe:

- the business domain model with its resources and attributes (section 2 & 3)
- the possible actions or commands (section 4)
- the new API definition, which is well aligned with the business domain model (section 5)
- other elements, which are yet to be described more in detail at the moment of writing (section 6)

Contents

- [1/ Introduction](#)
- [2/ Resources and their attributes](#)
 - [2.1/ Certificates](#)
 - [2.1.1/ Certificate attributes](#)
 - [2.2.2/ Certificate lifecycle](#)
 - [2.2/ Slices](#)
 - [2.2.1/ Slice attributes](#)
 - [2.2.2/ Slice lifecycle](#)
 - [2.3/ Wallets](#)
 - [2.3.1/ Wallets attributes](#)
 - [2.3.2/ Wallets lifecycle](#)
 - [2.4/ Claims](#)
 - [2.4.1/ Claims attributes](#)
 - [2.4.2/ Claims lifecycle](#)
- [3/ Business Domain Model](#)
- [4/ Actions or commands](#)
 - [4.1/ Atomic commands on a Slice](#)
 - [4.1.1/ createSliceAtomic](#)
 - [4.1.2/ transferSliceAtomic command](#)
 - [4.1.3/ sliceSliceAtomic command](#)
 - [4.1.4/ claimSliceAtomic command](#)
 - [4.1.5/ withdrawSliceAtomic command](#)
 - [4.1.6/ expireSliceAtomic command](#)
 - [4.2/ Higher level commands](#)
 - [4.2.1/ Issue a Certificate](#)
 - [4.2.2/ Transfer \(part of\) a Slice: transferSlice](#)
 - [4.2.3/ Claim \(part of\) Slices: claimSlice](#)
- [5/ API endpoints](#)
 - [Certificates](#)
 - [Slices](#)
 - [Wallets](#)
 - [Claims](#)
- [6/ Other elements](#)

2/ Resources and their attributes

In the Granular Certificate Registry domain, we can distinguish 4 resources:

- certificates
- slices
- wallets
- claims

2.1/ Certificates

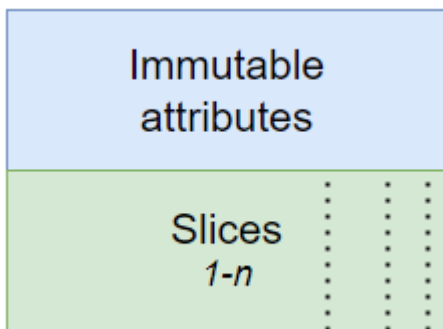
Granular Certificates are immutable objects that represent the produced or consumed energy by an Asset in a given timeperiod. A Certificate can describe either a production or consumption of energy of an Asset.

A Certificate:

- Has an immutable "header" which is the collection of attributes on the Certificate. These data cannot be changed after the Certificate has been issued.
- Has a collection of Slices linked to it. When a Certificate is issued, it has one initial Slice, representing the complete volume of the Certificate.

All commands on an existing certificate and the life-cycle happens through the slices.

Visual representation:



2.1.1/ Certificate attributes

A Certificate has a series of attributes. Some attributes only apply to Production type of certificates, hence we split in two categories.

Attributes on Production & Consumption certificates

- **certificateID**: Unique identifier of the certificate
- **registry**: Reference (URI) to the registry that holds the certificate
- **certificateType**: Type of certificate. Valid values are "production" or "consumption" (later to be extended towards storage types of certificates)
- **issued**: Date & time the certificate was issued, in ISO-8601 format and UTC
- **periodStart**: Timestamp of the start of the period in between the energy was produced/consumed, in ISO-8601 format and UTC (start is inclusive)
- **periodEnd**: Timestamp of the end of the period in between the energy was produced/consumed, in ISO-8601 format and UTC (end is exclusive)
- **quantity**: energy produced/consumed in the specified period, expressed as a whole number in Wh.
- **assetID**: unique identifier of the asset
- **assetCode**: the code by which the grid operator identifies the delivery point
- **assetCodeType**: the code type by which the grid operator identifies the delivery point.

Attributes only on Production certificates

- **techCode:** standard code to describe the technology used by the producing asset to produce energy. Codes are defined by AIB (Association of Issuing Bodies) in their factsheet 5.
- **fuelCode:** standard code to describe the fuel used by the producing asset to produce energy. Codes are defined by AIB (Association of Issuing Bodies) in their factsheet 5.
- **address:**

<code>streetCode</code>	<code>string</code>	The code of the street.
<code>streetName</code>	<code>string</code>	The name of the street.
<code>buildingNumber</code>	<code>string</code>	The building number.
<code>floorId</code>	<code>string</code>	The floor Id.
<code>roomId</code>	<code>string</code>	The room Id
<code>postCode</code>	<code>string</code>	The postal code of the building.
<code>cityName</code>	<code>string</code>	The name of the city
<code>citySubDivisionName</code>	<code>string</code>	The city subdivision name
<code>municipalityCode</code>	<code>string</code>	The municipality code
<code>locationDescription</code>	<code>string</code>	A description of the location.

- **location:**

<code>lat*</code>	<code>number(\$double)</code>	Latitude (the north-south position of a point)
<code>long*</code>	<code>number(\$double)</code>	Longitude (the east-west position of a point)

2.2.2/ Certificate lifecycle

No lifecycle/states are kept at Certificate level. The Slice(s) under a Certificate have their lifecycle/state. When a Certificate is issued, it has one initial Slice, which has the state "Active".

2.2/ Slices

As described above, a Certificate is immutable, and contains the entire quantity measured at the Asset's meter for the period, so how could transferring (trading) and claiming parts of a certificate work?

To solve this, a term from the financial sector was borrowed, stock-slicing, where a single stock can be traded as slices. This happens on top of the existing stock market.

A Certificate consists of 1 to n slices. When the Certificate is issued, one initial slice is created. Similar to baking a cake, "1 slice (whole cake)" exists.

A slice is always owned by a single Certificate Account.

2.2.1/ Slice attributes

A Slice has following attributes:

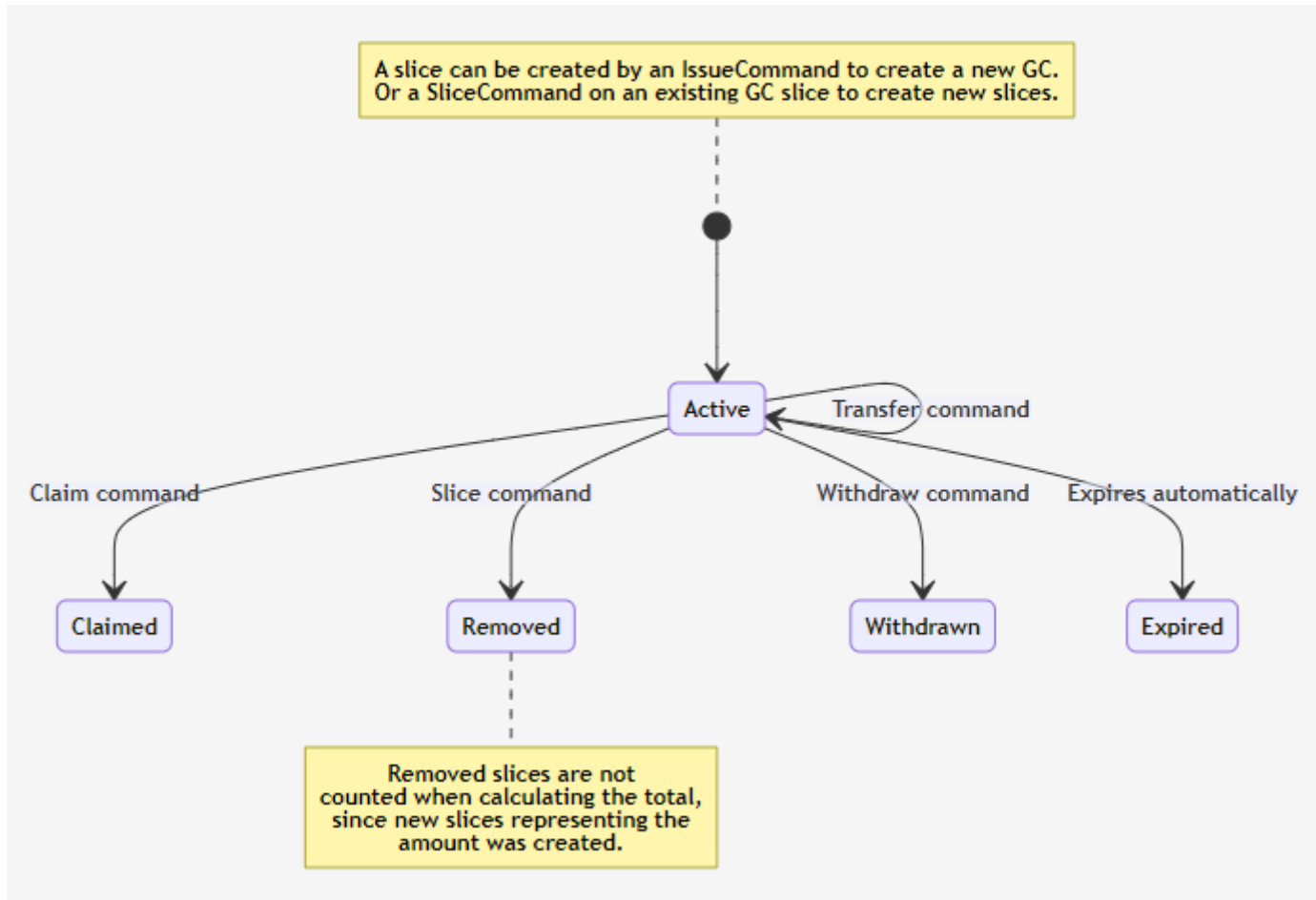
- **sliceID**: Unique identifier of the certificate
- **quantity**: energy represented by the Slice, expressed as a whole number in Wh.
- **sliceState**: Current state of the Slice. According to the lifecycle described above, following states are possible: *Active, Removed, Claimed, Withdrawn, Expired*
- **ownerWallet**: ID of the Wallet that holds the Slice
- **certificateID**: reference to the ID of the Certificate to which this Slice belongs.

2.2.2/ Slice lifecycle

A slice has a specific life-cycle. When the slice is created, it becomes active.

When a slice changes state, the state is stored on the slice.

Note that most commands are final, in there is no way to reverse them once performed.



2.3/ Wallets

Wallets are envelopes that hold Slices. The owner of a Wallet is owner of the Slices in that Wallet.

2.3.1/ Wallets attributes

Wallets have a public key and name. The public key is the unique identifier that will be used everywhere to identify the Wallet.

2.3.2/ Wallets lifecycle

Wallets have a state. The state is either 'active' or 'closed'.

- Wallets can't be deleted.
- Wallets can be closed if there are no active Slices in the Account. A closed Wallet can still contain non-active Slices. Once a Wallet is closed, Slices can no longer be transferred towards it.

2.4/ Claims

Through Claims, Certificate Slices get definitively linked to each other, and as such form the claim of the origin of consumed energy.

Each Claim is linked to exactly 2 Slices. Currently, we only allow Claims between Production Certificate Slices and Consumption Certificate Slices. (Will be extended later, when Storage Assets are introduced)

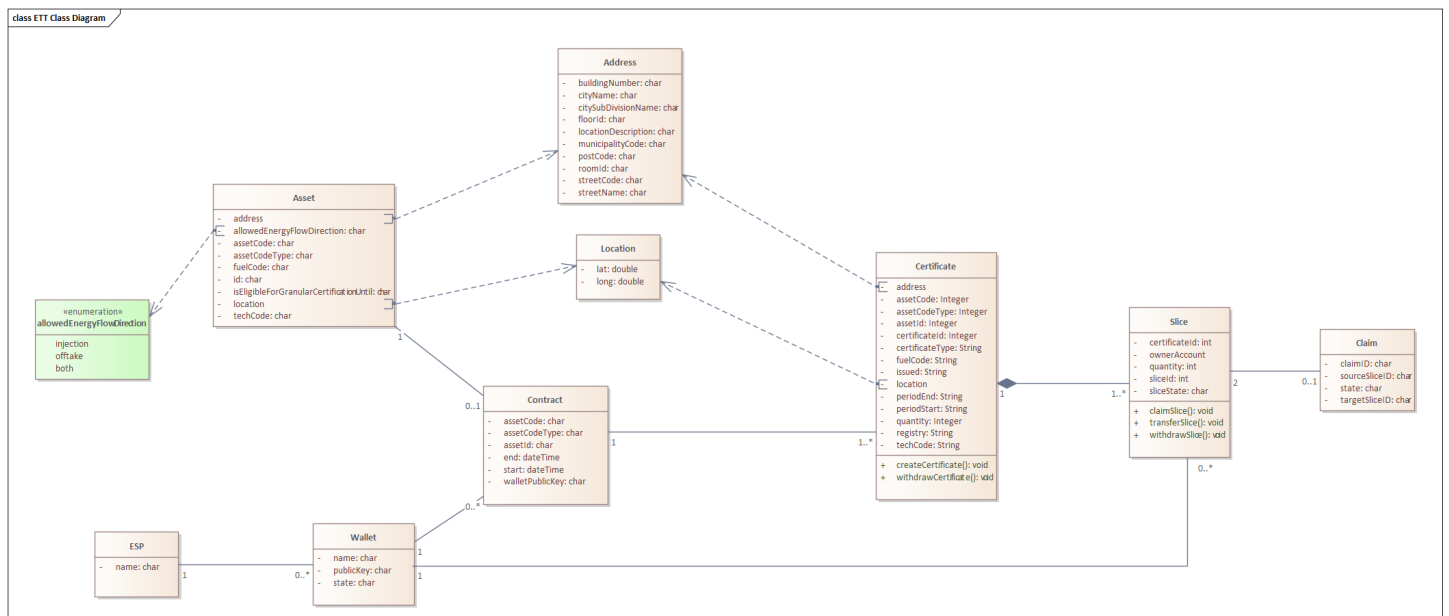
2.4.1/ Claims attributes

- claimID
- sourceSliceID
- targetSliceID

2.4.2/ Claims lifecycle

Claims have a state. At this moment, the state can only be 'validated'. As soon as a Claim is created, it receives the state 'validated'. Other states (and state change processes) will be defined later, when the proof layer, cross-border claims and GO integration are added.

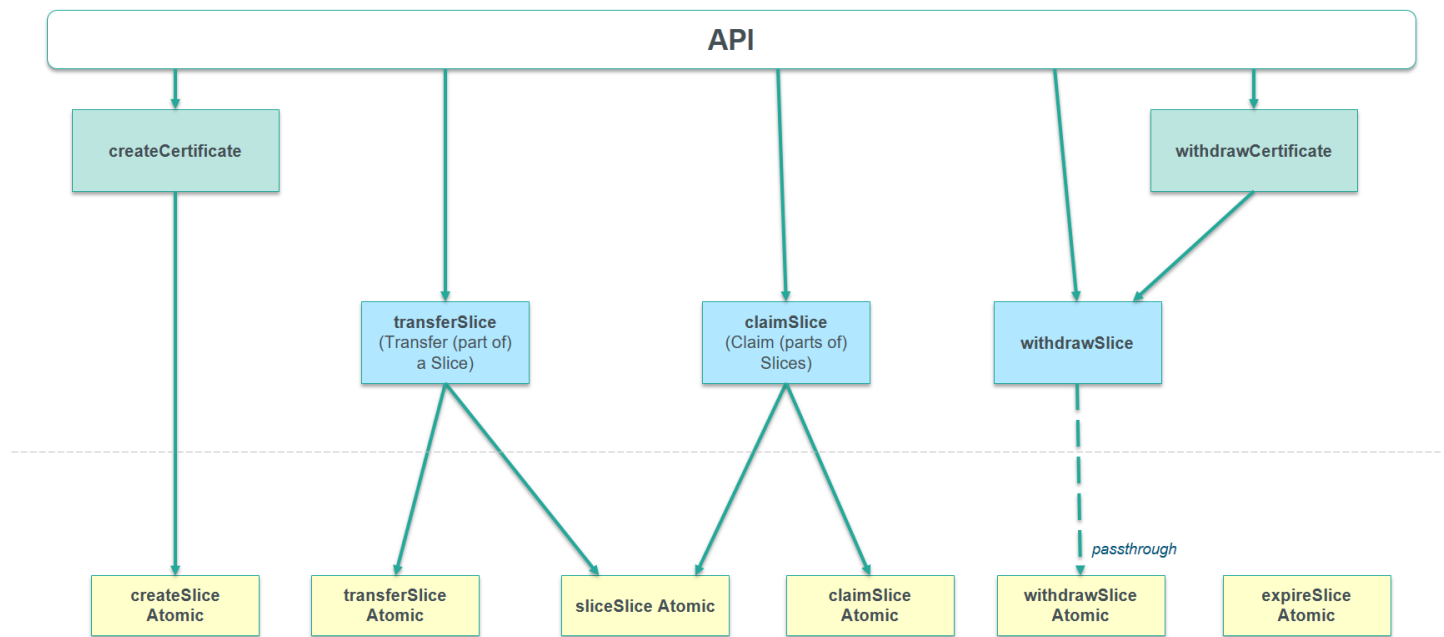
3/ Business Domain Model



4/ Actions or commands

Visual overview of the possible actions/commands. The commands are organized in multiple layers:

Actions/commands



4.1/ Atomic commands on a Slice

A number of atomic commands can be made on Slices.

We say 'atomic' because these are the most basic commands that can be made on Slices.

4.1.1/ createSliceAtomic

Slices can be created in two situations:

- Upon issuance of a Certificate: one linked Slice is created with a quantity equal to the Certificate's quantity, and referring to the Wallet specified by the issuance contract.
- Upon launching a **sliceSliceAtomic** command on an existing Slice to create two new Slices (see further: sliceSliceAtomic command)

The command **createSliceAtomic** will hence never be called directly, but will be called by the command **issueCertificate** (happening in the contract executer) or by the **sliceSliceAtomic** command (defined below).

Once a Slice is created, it gets the status "Active".

4.1.2/ transferSliceAtomic command

Transfers the ownership of an existing slice to a new certificate account (owner).

Inputs needed for this command:

- **sliceID**: ID of the Slice to be transferred
- **targetWallet**: ID of the Certificate Account that should receive the Slice.

The command will only succeed if:

- the specified Slice is in state "Active";
- the specified targetWallet is valid and "Active";
- the user that calls the command has permissions on the Slice (= the Wallet that currently holds the Slice).

Output in case of success:
Show all attributes of the Slice.

4.1.3/ sliceSliceAtomic command

The slice command enables the current owner to create new slices from any existing active slice. The new slices may be assigned to a new owner (=Wallet) as part of the slice command.

The sum of the new slices are always equal to the quantity of the old slice.

When the new slices are created, they become Active, and the old slice goes into state "Removed".

Inputs needed for this command:

- **sliceID**: ID of the originating Slice to be sliced.
- collection of new Slices to be created. Each item in the collection has following attributes:
 - **quantity**
 - **targetWalletPublicKey** (optional) : the Wallet that should receive the new Slice. If the targetWallet is not specified, that new Slice will be placed in the same Wallet as the originating Slice.

The command will only succeed if:

- the specified Slice is in state "Active";
- the specified targetWallets are valid and "Active";
- the user that calls the command has permissions on the Slice (= Wallet that currently holds the Slice).
- the sum of all quantities in the collection is **equal or less than** the quantity of the originating Slice.

If the sum of all quantities in the input collection **is less than** the quantity of the originating Slice, an extra new Slice will be created that has the remainder of the quantity, so that the **sum of all newly created Slices = the quantity of the originating Slice**. That last Slice will be placed in the same Certificate Account as the originating Slice. After all new Slices are created, the originating Slice goes into state Removed.

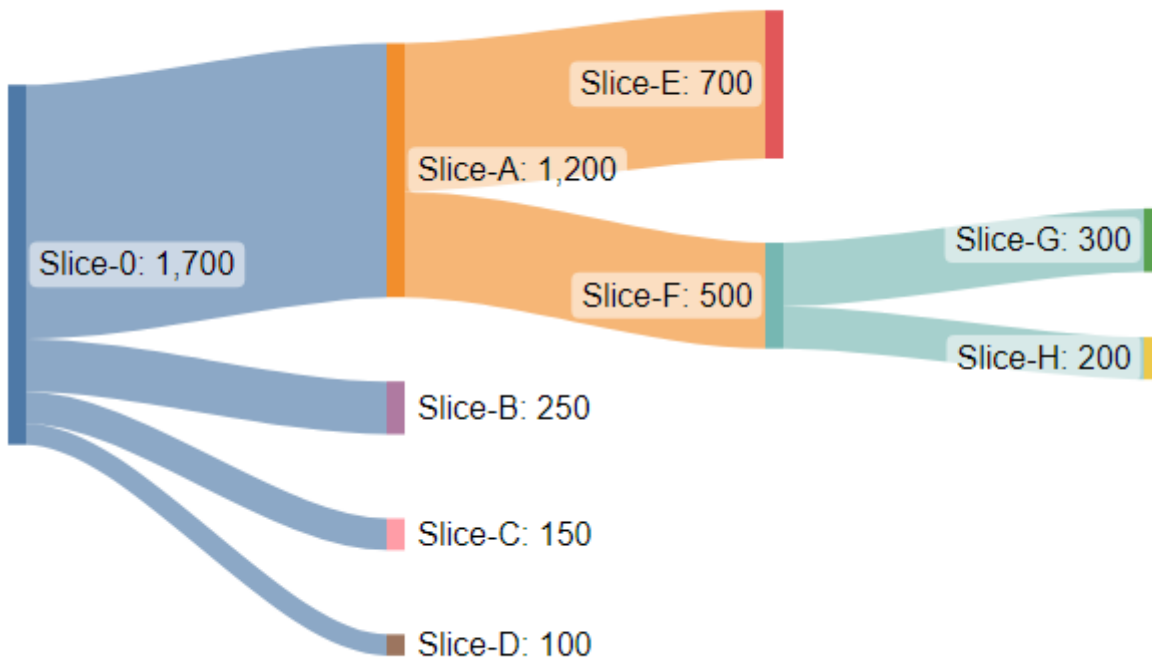
Output in case of success:
Collection of all newly created Slices, showing all attributes of each Slice.

Example of slicing:

Below is a Sankey representation of slice that is sliced multiple times:

Initially, a single slice (0) is issued with a quantity of 1700Wh.

- Slice-0 is sliced into 4 slices:
 - Slice-A 1200Wh
 - Slice-B 250Wh
 - Slice-C 150Wh
 - Slice-D 100Wh
- Slice-A is sliced into 2 slices:
 - Slice-E 700Wh
 - Slice-F 500Wh
- Slice-F is sliced into 2 slices:
 - Slice-G 300Wh
 - Slice-H 200Wh



Made with SankeyMATIC

At the end of the 4 Slice commands, the Slices have following states:

- Slice-0: Removed
- Slice-A: Removed
- Slice-B: Active - 250Wh
- Slice-C: Active - 150Wh
- Slice-D: Active - 100Wh
- Slice-E: Active - 700Wh
- Slice-F: Removed
- Slice-G: Active - 300Wh
- Slice-H: Active - 200Wh

Sum of all non-Removed slices is 1700Wh (=the quantity of the initial Slice = the quantity of the linked Certificate)

4.1.4/ claimSliceAtomic command

The Claim command is where an active slice linked to production certificate and an active slice linked to consumption Certificate get definitively linked to each other, and as such form the claim of the origin of consumed energy.

To use the claim command, one has to specify two Slices, one production and one consumption (*current rule, will evolve when Storage is added*), and they **have to be of equal quantity**.

This might require one to perform some slice commands first to make them fit together. (see higher level commands)

Once a claim has been created, it cannot be undone.

Inputs needed for this command:

- **sourceSliceID**
- **targetSliceID**

The command will only succeed if:

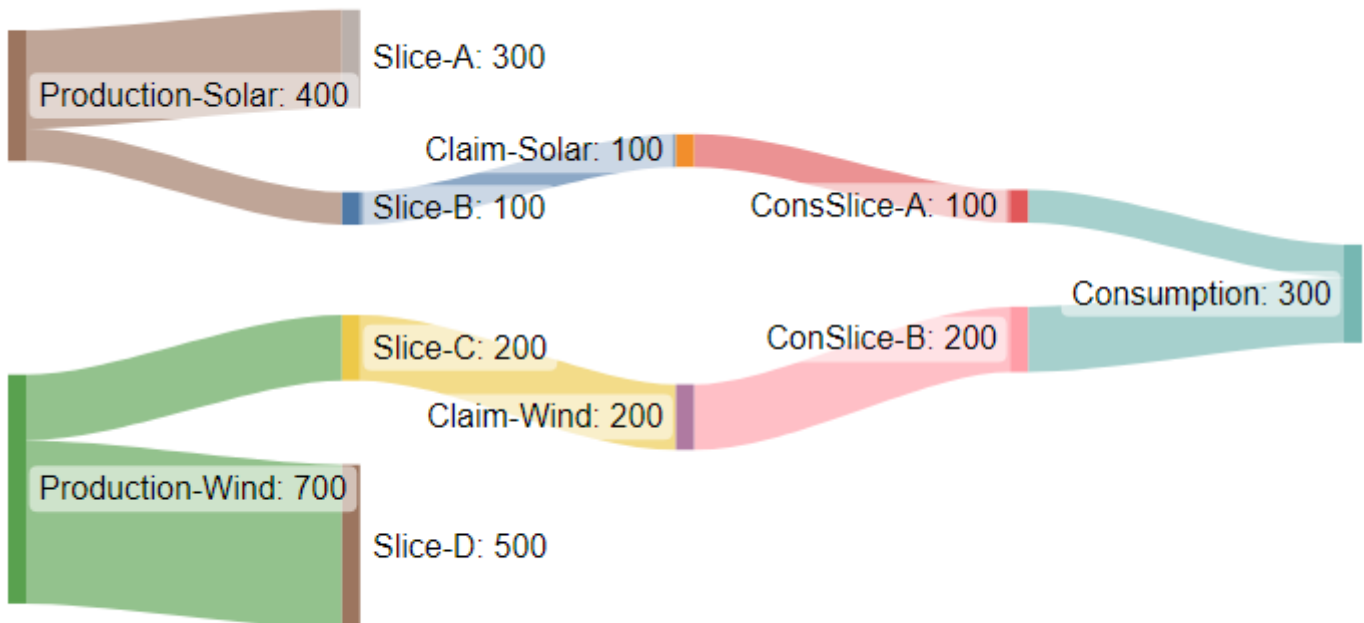
- both specified Slices are in state "Active";

- one Slice is linked to Certificate with type=production, and the other Slice is linked to Certificate with type=consumption (*current rule, will evolve when Storage is added*)
- both specified Slices have equal quantities.
- the user that calls the command has permissions on both Slices (= on the Wallet that hold the sourceSlice AND on the Wallet that holds the targetSlice).

Example of Slicing & Claiming:

In the example below 3 GCs exists, production-solar, production-wind and Consumption. A series of commands are then executed:

- Slice Consumption to ConSlice-A and ConSlice-B
- Slice Production-Solar to Slice-A and SliceB
- Claim Slice-B to ConsSlice-A
- Slice Production-Wind to Slice-C and SliceD
- Claim Slice-C to ConsSlice-B



Made with SankeyMATIC

After these 5 commands, the Slices are in following states:

- Production-Solar: Removed
- Production-Wind: Removed
- Slice-A: Active - 300Wh
- Slice-B: Claimed - 100Wh (against ConsSlice-A)
- Slice-C: Claimed - 200Wh (against ConsSlice-B)
- Slice-D: Active - 500Wh
- Consumption: Removed
- ConsSlice-A: Claimed - 100Wh (against Slice-B)
- ConsSlice-B: Claimed - 200Wh (against Slice-C)

So the Production Solar certificate of 400Wh has 300Wh still active, and 100Wh claimed against Consumption certificate.

The Production Wind certificate of 700Wh has 500Wh still active, and 200Wh claimed against Consumption certificate.

The Consumption Certificate of 300Wh is then fully claimed, against two energy sources: Solar & Wind.

4.1.5/ **withdrawSliceAtomic** command

An active Slice can be put in status "withdrawn" using this command.

4.1.6/ **expireSliceAtomic** command

Slice expire is not a command, but a rule which applies that all active slices on a certificate are invalid to perform commands on after a set amount of time.

Not to be implemented yet, as some business rules need further definition.

4.2/ Higher level commands

As described earlier, all actions or command on certificates will happen on Slice level.

Even though atomic actions are taken on Slice level (see prev section), this doesn't exclude the definition of higher level commands which then run a series of atomic command, to ease the user's interaction with the registry.

4.2.1/ **Issue a Certificate**

In the CCP contract service, the contract "GC issuance" will generate Certificates for a given Asset, based on received metering data, and put the initial slice in the specified Wallet.

For each period with metering data, the contract executer will run a command `issueCertificate`, which should generate the (Production or Consumption) Certificate.

In second order, every command "issueCertificate" that is run, should then run the command `createSliceAtomic`, to create the initial Slice that is linked to the Certificate.

The quantity of the initial Slice equals the quantity of the Certificate.

The Slice is put in the Certificate Account that is specified in the contract.

4.2.2/ **Transfer (part of) a Slice: transferSlice**

A user wants to be able to transfer a Slice either completely or partially. Therefore, a higher level command is required, which we shall call ***transferSlice***.

Following inputs are needed:

- the **sliceID** of the existing Slice which we want to transfer completely or partially. This Slices should be in state Active.
- the **quantity** of the Slice we want to transfer. This field is optional.
- the **targetWallet** : the public key of the Wallet that should receive the Slice (part).

Following scenarios can happen:

- If the specified quantity is omitted or equals the quantity of the originating Slice:
 - just execute the atomic command ***transferSliceAtomic*** .
- If the specified quantity is less than the quantity of the originating Slice:
 - execute the atomic command ***sliceSliceAtomic***, specifying following collection in the input body:
 - Slice with the quantity as specified in the command. This slice goes into the specified targetWallet.
- If the specified quantity is larger than the originating Slice: return an error and don't apply any transfer.

4.2.3/ **Claim (part of) Slices: claimSlice**

A user wants to be able to claim two Slices with different sizes, or wants to specify the quantity to be matched. therefore, a higher level command is required, which we shall call ***claimSlice***.

Following inputs are needed:

- **sourceSliceID**
- **targetSliceID**
- **quantity**: the quantity to be claimed (optional)

The command will only succeed if:


- the specified quantity is $<$ or $=$ to $\text{MINIMUM}(\text{quantity of productionSlice}, \text{quantity of consumptionSlice})$
- the underlying atomic commands succeed.






Following scenarios can happen:

- if no quantity is specified, or the specified quantity $<$ or $=$ $\text{MINIMUM}(\text{quantity of productionSlice}, \text{quantity of consumptionSlice})$:
 - call atomic command **sliceSliceAtomic** on the productionSliceID, and providing the specified quantity;
 - from the response, save the sliceID of the new slice that has the specified quantity.
 - call atomic command **sliceSliceAtomic** on the consumptionSliceID, and providing the specified quantity;
 - from the response, save the sliceID of the new slice that has the specified quantity.
 - call atomic command **claimSliceAtomic**, specifying the two saved sliceIDs from previous steps.
- if the specified quantity is $>$ than $\text{MINIMUM}(\text{quantity of productionSlice}, \text{quantity of consumptionSlice})$:
 - throw an error, stating that "The specified quantity to be matched is larger than the smallest Certificate Slice".

5/ API endpoints

Certificates

certificates Granular Certificates are immutable objects that represent the produced or consumed energy by an Asset in a given timeperiod. 

GET	<code>/interface/registry/v1/certificates</code>	Get Granular Certificates in the registry	  
POST	<code>/interface/registry/v1/certificates</code>	[Issuing Body only] Create Granular Certificate in the registry	
GET	<code>/interface/registry/v1/certificates/{certificateID}</code>	Get Granular Certificate with specified ID	
DELETE	<code>/interface/registry/v1/certificates/{certificateID}</code>	[Issuing Body only] Withdraw Granular Certificate with specified ID	

Slices

slices

A Granular Certificate has a collection of Slices linked to it. When a Granular Certificate is issued, it has one initial Slice, representing the complete volume of the Certificate. This Slice can be further split, and Slices can be transferred from one Wallet to another. ^

GET `/interface/registry/v1/slices` Get Slices in the registry v

GET `/interface/registry/v1/slices/{sliceID}` Get Slice with specified ID v

DELETE `/interface/registry/v1/slices/{sliceID}` [Issuing Body only] Withdraw a Slice with specified ID v

POST `/interface/registry/v1/slices/{sliceID}/transfer` Transfer (part of) a Slice to another Wallet. v

POST `/interface/registry/v1/slices/transfer` Transfer a collection of Slices (partially) to another Wallet. v

Wallets

wallets Wallets are envelopes that hold Slices. The owner of a Wallet is owner of the Slices in that Wallet.

GET `/interface/registry/v1/wallets` Get Wallets

POST `/interface/registry/v1/wallets` Create new Wallet

GET `/interface/registry/v1/wallets/{publicKey}` Get Wallet with specified Public Key

PATCH `/interface/registry/v1/wallets/{publicKey}` Update Wallet

DELETE `/interface/registry/v1/wallets/{publicKey}` Close a Wallet

Claims

claims

Through Claims, Certificate Slices get definitively linked to each other, and as such form the claim of the origin of consumed energy.

GET `/interface/registry/v1/claims` Get Claims

GET `/interface/registry/v1/claims/{claimID}` Get Claim with a specified ID

POST `/interface/registry/v1/claims/slices` Make Claim between Slices

POST `/interface/registry/v1/claims/advanced` Make Claims based on advanced criteria

6/ Other elements

Other elements under consideration, to be further developed:

- Technical performance measures:
 - GET endpoints: foresee *pagination/filtering/sorting* to keep things performant
 - Bulk transfers & claims: turn into an *async operation* -> Return URI where to check status/outcome of the request
- GET Certificates or Slices: possibility to include info about the claim, if a claim was made on that Slice.
- Add a fifth endpoint group: **/reports**
 - Return indicators or timeseries that give insights into origin of energy, based on certificates, slices, claims made.
 - Timeseries that shows consumption by fuelcode, techcode, asset