TOMP- API v2.0

Blueprint 2-wheeled shared vehicles

[Foreword 3](#_Toc201736057)

[Objectives 3](#_Toc201736058)

[Summary 3](#_Toc201736059)

[Context 4](#_Toc201736060)

[Other TOMP blue prints 4](#_Toc201736061)

[Terms and Definitions 5](#_Toc201736062)

[Roles & architecture 5](#_Toc201736063)

[Design principles 6](#_Toc201736064)

[Migration strategy 7](#_Toc201736065)

[Flow overview 8](#_Toc201736066)

[Offer flows 9](#_Toc201736067)

[Asset based 9](#_Toc201736068)

[Offer based 10](#_Toc201736069)

[Product based 11](#_Toc201736070)

[Purchase flows 12](#_Toc201736071)

[Immediate confirmed 12](#_Toc201736072)

[Auto-confirm 13](#_Toc201736073)

[2-phase purchase 13](#_Toc201736074)

[Execution flows 14](#_Toc201736075)

[Support flows 16](#_Toc201736076)

[Payment flow 17](#_Toc201736077)

[Deposits 17](#_Toc201736078)

[Pay when finished 17](#_Toc201736079)

[Subscriptions 17](#_Toc201736080)

[After sales flow 18](#_Toc201736081)

[Redresses 18](#_Toc201736082)

[Special cases 18](#_Toc201736083)

[Appendix A: functions 20](#_Toc201736084)

[Appendix B: OGC Meta data 25](#_Toc201736085)

[Appendix C: OGC structure 25](#_Toc201736086)

[Appendix D: Authentication 27](#_Toc201736087)

[Appendix E: Authorisation 29](#_Toc201736088)

[Appendix F: Pagination 30](#_Toc201736089)

[Copyrights 30](#_Toc201736090)

# Foreword

The aim of this document is to describe the profile for 2-wheeled vehicles of the TOMP-API. The TOMP-API provides a wholistic API for all modes, and requires profiles per mode to make it understandable and easy to implement.

# Objectives

This document is written for these audiences:

* Business developers
* Business architects
* Software developers

The aim is to describe the process served by the API, make it relatable to your business process and to make it implementable.

# Summary

The TOMP-API includes the complete process of the user journey on 2-wheels (or a chained trip containing a part on 2 wheels).

The process is divided into several modules, each has its own flow.

The ***offer flow*** for 2-wheeled vehicles has 3 different starting points:

1. Start with a vehicle on the map, on the street or in external data sources
2. Start with search for offers (based on journey planning)
3. Purchase additional products (subscriptions, day-cards etc.).

The ***purchase flow*** has 2 flavors:

1. Immediately confirmed, but with a ‘cancel window’ to rollback without financial consequences
2. A 2-phase purchase, where it is required to execute a second step, to confirm the purchase

The ***execution flow*** is pretty unique for 2-wheeled vehicles. It is because the traveller is in control, instead of someone else. It contains (amongst others) the steps of starting, ending, pausing, resuming the vehicle.

This part describes also the assignment of assets, and ancillaries.

The ***support flow*** is needed if you want to support unusual situations, like flat tires. To implement this, the reseller needs to implement the *notification module,* since it uses call-backs.

The ***payment flow*** can be used to report the balance, but also to request deposits. Optionally, it can be used to request direct payments. This part can be found in the *after-sales module.*

There are a few other modules, like the *customer registration module*, the *pre-sales module*, the *technical module* and the *discovery module*, but these are less relevant from the process perspective.

# Context

{MaaS levels of maturity}

Afbeelding met tekst, schermopname, Lettertype, nummer

Door AI gegenereerde inhoud is mogelijk onjuist.

{explain usage of GBFS and NeTEx in combination with the TOMP processes}

# Other TOMP blue prints

We’ll produce later on other blue prints, but we already can describe the major differences:

* Public transport (train, tram, bus): the execution and support part are less relevant, the refund-deposit is absent. Things like products, offers and seat locations are specific for this domain. Also the ‘pre-sales’ part is important: tailoring the offer(s) into a single package.
* On-demand cars/taxi: the notification module (MP side) is important, since the ride will be controlled by a (human) driver. Notifications like ETAs are not very common in other domains.
* Shared cars: {Roman}
* Parking locations: the traveller is often defined by a license plate, the ‘execution’ phase is often already started, the purchase/payment is attached later on, often just before leaving the facility. Specific operations: opening entrance side of the building. On-street parking on the other hand requires notifications (like an approaching end time)
* 2-wheeled vehicles: especially the execution part is import here. Communication of (non)-parking areas, notification of end-times, out-of-zones, but also publishing operational instructions (how to start, open the helmet box, etc.).

Due to these differences there a certain need to have a ‘profile’ per mode. This blue print describes the profile for the 2-wheeled vehicles.

# Terms and Definitions

The **API**: the Transmodel based REST API this document describes.

**Asset:** in this context, normally a 2-wheeled vehicle (like the bike, scooter or step). It can be used to execute a single leg.

**Booking:** Due to the ambiguity we avoid the term booking. A booking is realized through creating and purchasing a package.

**Package:** A container to transmit the data of legs, products (‘asset types’) and conditions. In more complex contexts, it can contain also offers covering parts of the complete package. In the 2-wheeled context, a package contains 1 offer and it is not mandatory to communicate this.

**Leg**: A (part of a) trip, that can be made using one asset. In rare situations, a second asset can be assigned to the same leg (e.g. in case of a replacement).

**Customer:** the person or organization that purchases the package. It is not required that the customer travels as well.

**Traveller**: An individual or group using the transport services. In case of anonymous travelling, only a user profile is used. In this context, often a driver’s license is required, which requires to communicate individual travellers.

**Product**: In this context, an ‘asset type’, like a ‘normal bike’, ‘e-bike’ or ‘cargo bike’, not referring to a specific asset, but to it’s ‘class’. In some advanced implementations, it could also be a subscription (day pass).

**Offer**: an offer to use an asset, in most cases only containing a starting location, price (in case of fixed prices), a reference to the asset (or asset type). It must available at the specified location.

**Travel Document**: a – in most cases – technical proof to use an asset. It could be a only call to the TO, that can open the bike, a certificate, NFC, or Bluetooth information provided by the TO to open the bike.

**Ancillary**: a secondary product that can be added to the leg/asset. For instance, a bike helmet, or a child’s seat.

# Roles & architecture

**Customer**  
See explanation under Terms and Definitions.

**Traveller**  
See explanation under Terms and Definitions.

**MaaS Provider *(MP)***  
books on behalf of the customer and offers the end user experience to the customer. Also known as reseller.

**Aggregator / Distributor**  
who provides the mobility services on behalf of the Transport Operator(s). Could also be known as a distributor.

**Transport Operator *(TO)***organization who provides a flexible offering of transportation services. Also known as Mobility Providers or Transport providers (public transport context).  
Examples: scooters, bikes or car sharing.

**Identity Provider *(IdP)***  
verifies identity and provides authentication of customers for the reseller

**Payment Service Provider *(PSP)***  
handles payment processing between customer and MP, sometimes even between customer and TO.

Afbeelding met Lettertype, diagram, ontwerp

Door AI gegenereerde inhoud is mogelijk onjuist.

The MP (depicted as mobile phone) is responsible for route planning, finding the aggregators or TOs, and contact the ones who are possibly applicable to provide a leg to serve the customer’s travel demands.

The API can be used between:

* MP and aggregators/distributors
* Aggregator/distributor and TOs
* Amongst aggregators/distributors

# Design principles

A short overview of design principles we took into account:

* *“Don’t break the bank”*
  + Short return on investment
  + The implementing party is in control, describing what it has implemented and how. This allows to extend the implementation over time.
* *“Proactive error prevention”*
  + Easy to understand and use, be intuitive
  + Provide meta data for all functions
  + Make clear what is possible at any time (RFC 5988)
  + Provide context per returned entity (e.g. ‘this is an offer’)
  + Don’t patch data, we use logical operations. Patch requires external parties to know internal business logic.
  + Meta data must be provided in human AND machine readable format
  + Operations always return the created/modified resource, not parts of it
  + Be clear about technical aspects, like identification, authenticity and authorisation (certificates, tokens)
  + Be clear about versions (semantic versioning).
* *“Don’t redo what others have done already”* - Reuse data sets that are already available (externally, like in NAPs)
* *“Mobility for all”* - Provide an API that is usable for all modes
* *“There is a lot of knowledge out there”* - Comply to existing technical (meta) standards, techniques and ontologies
  + Transmodel/NeTEx
  + OGC API standards
  + RestFULL API
  + OpenAPI 3
  + (Geo)JSON
  + OAuth2, JWT, PKI
  + RFC’s, for e.g. date formats, problem details, language and country specifications
  + Verifiable credentials/presentations

When applying these principles, there are quite a few technical problems already solved. In the next chapter, technical aspects are enlisted, and quite a few are referring back to the list above.

# Migration strategy

{some text how to cope with other versions, or existing APIs}

# Flow overview

Afbeelding met tekst, diagram, lijn, Plan

Door AI gegenereerde inhoud is mogelijk onjuist.

{probably need some text here}

# Offer flows

In each of the flows described below, we use a blue references in italics, they refer to the appendix, where more details are provided.

In short, you should ponder which flows you want to implement, probably not all are relevant.

## Asset based

This is a pretty common scenario. You get data for available bikes, scooters, etc., from a source (could be NeTEx, GBFS, or another standard), or maybe it's just right in front of you, on the street. You provide the ID (you can find references using the datasources, or it's the visual ID on the vehicle), and the MP can request to use this asset (use-asset).

Afbeelding met Mobiele telefoon, gadget, Elektronisch apparaat, Communicatieapparaat

Door AI gegenereerde inhoud is mogelijk onjuist.

NeTEx

GBFS

The response comes back in either a PENDING or CONFIRMED state (check out the *purchase flows*). In both cases, the required resources are claimed.

Afbeelding met tekst, schermopname, lijn, Lettertype

Door AI gegenereerde inhoud is mogelijk onjuist.

This function allows to book an asset with only one call! It returns a purchased package with a reference (deeplink) into your app, making it very easy to integrate into other solutions.

## Offer based

Another option is to search for offers based on start location, timestamp, and other user requirements (search-offers). These offers are NON-BINDING, meaning the resources aren’t claimed yet.  
  
You can then select a single offer and purchase it (purchase-offers).

Afbeelding met tekst, schermopname, lijn

Door AI gegenereerde inhoud is mogelijk onjuist.

The response from the purchase-offers call will be in either a **PENDING** or **CONFIRMED** state (see *purchase flows*). In both cases, the necessary resources are claimed.

Afbeelding met tekst, Mobiele telefoon, gadget, Draagbaar communicatietoestel

Door AI gegenereerde inhoud is mogelijk onjuist.If you’re allowed to modify offers (common in things like public transport or shared cars), or if you need to provide binding offers, you can use the pre-sales module.

However, since this isn't common in this context, it’s not covered in this blueprint.

## Product based

Afbeelding met tekst, schermopname, Lettertype, lijn

Door AI gegenereerde inhoud is mogelijk onjuist.

Often, bike operators sell products like weekly cards, day cards or cards with e.g. 10 rides. These products can be sold using purchase-product.

Afbeelding met tekst, gadget, Mobiele telefoon, Draagbaar communicatietoestel

Door AI gegenereerde inhoud is mogelijk onjuist.

The returned package is default in a CONFIRMED state, but it could be PENDING depending on the business case (see *purchase flows*). These purchased products normally don’t have consequences for claiming resources, but can later on be used in the offer flows (asset based or offer based), as cards.

# Afbeelding met tekst, Mobiele telefoon, gadget, Communicatieapparaat Door AI gegenereerde inhoud is mogelijk onjuist.Purchase flows

At this stage, a service in the shape of an asset or product has been selected (the state is PENDING or CONFIRMED). We are facing 3 different flows:

* immediate confirmed
* auto confirm and
* the 2-phase purchase

## Immediate confirmed

The returned purchase has the status ‘CONFIRMED’. Until the timestamp in the ‘Expiry-date’ header field, the purchase can be undone using the rollback-purchase. This is a requirement, although the TO can specify itself how long it will give the opportunity to roll back the purchase without financial consequences.

After this timestamp, it can only be done using the refund-options, or in case of a technical problem, using cancel-package.

Afbeelding met tekst, schermopname, lijn, Lettertype

Door AI gegenereerde inhoud is mogelijk onjuist.

## Auto-confirm

The auto-confirm flow is more or less the same as the ‘immediate confirmed’-flow, except that the returned state is ‘PENDING’ until the timestamp specified in the expiry-date field has passed.

Afbeelding met tekst, lijn, schermopname, diagram

Door AI gegenereerde inhoud is mogelijk onjuist.

In this flow, it is also possible to request an extension of the expiry time.

## 2-phase purchase

Afbeelding met schermopname, tekst, lijn, diagram

Door AI gegenereerde inhoud is mogelijk onjuist.

The 2-phase purchase starts with a package in the PENDING state. The purchase-offers, use-asset and purchase-product, the returned information contains an expiration date.

If the PENDING package is not confirmed (confirm-purchase) and there is no request to extend the expiration time, it will automatically end in an EXPIRED state.

If the MP (on behalf of the customer) does not want to continue with the purchase process, it can roll back the purchase (rollback-purchase), thereby releasing the required resources.

# Execution flows

We have two levels of flows here. One on the package level, and the other on the leg level. The package level flow is straight forward, you have a package, you can modify the package (like assigning assets, or add ancillaries like helmets), and off you go.

At the end, when all legs are ended, the package ends up in the state ENDED.

Afbeelding met tekst, diagram, schermopname, lijn

Door AI gegenereerde inhoud is mogelijk onjuist.

At leg level, there are more options. These leg flows can only be activated when the package is in STARTED state.

When the leg is in NOT STARTED state, it is possible to assign assets and ancillaries. This could impact the total price.

Afbeelding met tekst, schermopname, Lettertype, lijn

Door AI gegenereerde inhoud is mogelijk onjuist.Once a TO gets a request to start a leg (start-leg), it is possible that the TO needs some time to prepare the applicable asset related to the leg. It will return a leg in the state ‘PREPARING’. For this flow, the *notification module* must be implemented at the MP side, so the TO can inform the MP that the asset is prepared.

Afbeelding met tekst, elektronica, gadget, Elektronisch apparaat

Door AI gegenereerde inhoud is mogelijk onjuist.But, in most cases, when starting a leg, it will return a leg ‘in use’. The asset is unlocked and ready to use. In some occasions, the asset is not unlocked, but must be unlocked manually.

To perform operations directly on the asset, there is an operation-asset facility, to e.g. open a trunk, or to unlock a side door in a parking garage. This is not described in this blueprint, but it’s there.

Once the asset is in use, it can be paused (pause-leg) and resumed (resume-leg) again, or when needed, and the end time of leg is approaching (extend-leg), you can request to extend the leg as traveller.

Afbeelding met tekst, diagram, schermopname, lijn

Door AI gegenereerde inhoud is mogelijk onjuist.Ending the leg (end-leg) implies that the asset is locked and the financial calculations can be made.

# Afbeelding met tekst, persoon, gadget, Mobiele telefoon Door AI gegenereerde inhoud is mogelijk onjuist.Support flows

The support flow can only be activated per leg, when it is in execution. From any state within the leg-flow, an issue can be reported, and it has to be handled appropriately by the TO.

The MP is of course in charge of issues related to the app and the communication between the TO and MP, but when it relates directly to the asset, the MP has to rely on the TO to fix the issue.

Afbeelding met tekst, schermopname, diagram, lijn

Door AI gegenereerde inhoud is mogelijk onjuist.The help desk or support officer from the TO has multiple options, like fixing things at location, assign another asset to the leg (e.g. a replacing bike) or request an abnormal end of the leg. In this case, the after sales module (requiring refunds) facilitates the process.

# Payment flow

In this context, it is likely that the TO uses deposits, a kind of financial guarantee. For instance, when the MP and TO don’t know each other, it is a normal way of working.

## Deposits

Afbeelding met tekst, diagram, schermopname, lijn

Door AI gegenereerde inhoud is mogelijk onjuist.

Once the payment has been confirmed (and the TO can of course validate it on its bank account), the purchase can be returned. This approach can also been taken when the MP does not want to pay the deposit; it can relay the payment request to the customer.

This approach requires the request-payment (in the notification module) and the confirm-payment.

## Pay when finished

Beside upfront payments and deposits, we also know organizations that request immediate payment when the leg has ended. The same approach can be applied.

## Subscriptions

The majority of the implementations uses nowadays subscriptions. It is required to publish somehow, as a TO, how much credits you have from a certain MP. The payments can be used for this, so the MP can validate their own registration with the TOs.

# Afbeelding met tekst, schermopname, Lettertype, lijn Door AI gegenereerde inhoud is mogelijk onjuist.After sales flow

## Redresses

The after sales module contains, besides the payments, also the redress option. A redress can be a refund (‘money back’) or a replacement (another package, maybe valid on another day, or a week-card, same trip somewhat later including a cup of coffee, … )

Supplying the legs, ancillaries or package you want to request redresses for, will return redress options. If applicable, the TO returns valid redress options, it can contain financial compensations or replacements.

# Special cases

There are a few special cases (in comparison to other modes) in this blue print:

* blue tooth locks
* manual locks
* (non)-parking and other zones
* return areas or return stations
* open helmet-boxes
* communication of instructions
* warnings to the traveller

We’ll describe these in just a few lines per case.

### Blue tooth locks

Afbeelding met tekst, gadget, Mobiele telefoon, fiets

Door AI gegenereerde inhoud is mogelijk onjuist.Blue tooth locks are not standardized at all. This makes it hard to communicate with them. We see things like SDKs, but this limits the usage to the apps who want to incorporate the SDKs. And if you want to use many of these, your MP app will become very hard to maintain and very big. We don’t have a straightforward answer to this one.

### Manual locks

These locks require instructions, and those can be supplied using the ‘links’ part of the leg. A simple leg referencing to an instruction page on the internet, with a ‘required’ : true field, would force the MP to show the instructions. Often, the bike will communicate to its back-office that it has been opened, which can trigger a notification to the MP.

### Afbeelding met tekst, Mobiele telefoon, gadget, Elektronisch apparaat Door AI gegenereerde inhoud is mogelijk onjuist.(Non)-parking and other zones

During the ‘execution’ phase, the operator can choose to deliver geoJSON, including features referencing to e.g. MDS policy areas.

### Return areas or return stations

The same applies to return areas or stations, the stations or return areas can be represented as features, referring to GBFS items.

### Open helmet-boxes / custom asset operations

It is possible and allowed to ‘invent’ new endpoints, as long they comply to OGC processes. The flow can be altered, because you – as TO – prescribe the following steps in each returned concept (like a leg).

This could result in a POST /processes/open-helmet-box/execute, and the start-leg could specify a ‘open-helmet-box’ link. When this URL is called, the helmet box opens and in the result there is a ‘unlock-asset’ link and optionally a link with instructions.

### Afbeelding met tekst, motorfiets Door AI gegenereerde inhoud is mogelijk onjuist.Communication of instructions

In e.g. scooter solutions, it is very common to give instructions to operate the scooter, before each step. It is possible to add ‘instruction’ links in the link set, and you could even mark them as ‘mandatory’, to indicate the MP to show the website (or any other link) before the next link is executed.

Afbeelding met tekst, gadget, fiets, Draagbaar communicatietoestel

Door AI gegenereerde inhoud is mogelijk onjuist.

### Warnings to the traveller

This functionality must be implemented by the MP. It can be found in the notification module, and the TO can send notifications of the type WARNING or INFORMATION (or even more detailed notifications). The destination should be ‘TRAVELLER’.

# Appendix A: functions

search-offers

**Endpoint:** POST /process/search-offers/execute

**Functionality**: find offers, based on start location & time, optionally end location & time, general specification for all travellers & customer, detailed specifications of the travellers.

**Parameters**: travel details (from, to, times), traveller characteristics and traveller requirements

**Returns**: a collection of offers, each offer contains a summary and additional links to

1. Afbeelding met tekst, schermopname, Lettertype, diagram

   Door AI gegenereerde inhoud is mogelijk onjuist.details (calling GET /collections/offers/items)
2. purchase of this offer (calling POST /processes/purchase-offer/execute)

Example (pseudo-json, italics = optional):

{ **type**: offerCollection,  
 **offers**: [ { **type**: offer,   
 **id**: 1,   
 **properties**: { <see next example>},   
 *geometry: { … }*,   
 **links**: [ { **rel**: details, **href**: /collections/offers/items?offerId=1 }  
 , { **rel**: purchase: **href**: /processes/purchase-offers/execute  
 **method**: POST, **body**: { … } } ] } ] }

offers

**Endpoint:** GET /collections/offers/items?offerId=*[offerId]*

**Functionality**: retrieve the details of an offer, including the products and legs.

**Returns**: package containing one offer, including links to:

1. purchase of this offer (calling POST /processes/purchase-offer/execute)

**Example**:

{ **type**: offer,   
 **id**: 1,   
 **properties**: { **type**: offer,   
 **status**: OFFERED,   
 **from**: GPS:5.234,52.3432,  
 **startTime**: 2025-17-06T08:45:00.000Z,  
 **products**: [ { **type**: product, **id**: simple bike,  
 **fare**: { currencyCode: EUR,  
 elements: [ { **amount**: 1.50, **type**: FIXED },  
 { **amount**: 0.50, **type**: FLEX,  
 **units**: MINUTE, **amountOfUnits**: 15 } ] } }   
 ] },   
 *geometry: { … }*,   
 **links**: [ { **rel**: details, **href**: /collections/offers/items?offerId=1 }  
 , { **rel**: purchase: **href**: /processes/purchase-offers/execute  
 **method**: POST, **body**: { … } } ] }

Afbeelding met tekst, ontvangst, diagram, Parallel

Door AI gegenereerde inhoud is mogelijk onjuist.The ‘from’ and ‘to’ can be specified as GPS locations, but also refer to external data, like GBFS stations of NeTEx StopPoints. E.g. DKA:Station:342.   
  
To use these references, you have to include the referenced data source in the datasources. The example shown must include a link with rel ‘DKA:Station’, pointing to a GBFS station file (href) and with type ‘application/gbfs\_station\_information+json’.

purchase-offers

**Endpoint**: POST /processes/purchase-offers/execute

**Functionality**: purchase (aka ‘book’) directly from an offer (in this context it is normally only one offer)

**Parameters**: a reference to the offer, and where needed, additional traveller details (like a driver’s license number).

**Returns**: a purchased package, containing one offer. The endpoint returns also a package object, the same as the previous endpoint.

For advanced cases, where you can purchase multiple offers at once, there is an ‘offers’ fields (array), where you can find the individual offers (containing references to legs and products in this package).

**Example**: see previous example, but the status will be ‘PENDING’ or ‘CONFIRMED’, depending on the chosen flow: auto confirm/2-phase or immediately confirmed.

The ‘links’ collection contains

1. in case of the *auto-confirm flow* a rollback-purchase link, until the status has changed to ‘CONFIRMED’.
2. In case of the *2-phase purchase flow* both the rollback-purchase and the commit-purchase link
3. In case of the *immediately confirmed flow* it must contain a rollback-purchase link, until the expiry date has expired.

commit-purchase

**Endpoint**: POST /processes/commit-purchase/execute

**Functionality**: This function finalizes the purchase; the resources are claimed. If the customer wants to undo the purchase, it has to go through the ‘refund-options’.

**Parameters**: a reference to the package

**Returns**: a package with status ‘CONFIRMED’. The links collection SHOULD contain a ‘refund-options’ link and a ‘cancel-package’ (for technical errors).

rollback-purchase

**Endpoint**: POST /processes/rollback-purchase/execute

**Functionality**: This allows the customer to undo the purchase. It is only allowed to execute it when the package is in ‘PENDING’ state (auto-confirm or 2-phase purchase) or in ‘CONFIRMED’ state (immediate confirmed) AND before the expiry date.

**Parameters**: a package reference

**Returns**: a package with state ‘ROLLBACK’, this is a final status, so the link collection MUST be empty.

extend-expiry-time

**Endpoint**: POST /processes/extend-expiry-time/execute

**Functionality**: This optional endpoint allows the PM to request additional time to complete processes, like settlement of payment or interacting with other transport operators.

**Parameters**: a package reference

**Returns**: the same package, with an extended expiration timestamp. It is up to the TO how much time will be granted.

GET /collections/assets/items

POST /processes/assign-asset/execute

**Endpoint**: GET /collections/assets/items  
 & POST /processes/assign-asset/execute

**Functionality**: Assign a(nother) asset to a leg, or remove it from a leg

**Parameters**: a package & leg reference, an asset to assign to it, a replacing asset or an asset to remove.

**Returns**: A modified package, with an asset (bike, scooter) assigned to a leg. Normally, a single leg has a single asset assigned, but there could be complicated situations where a second asset is assigned (a replacing bike due to malfunctioning).

ancillaries   
  
assign-ancillaries

**Endpoint**: GET /collections/ancillaries/items   
 & POST /processes/assign-ancillaries/execute

**Functionality**: When package is confirmed, and additional ancillaries (not yet in the purchased package) must be added, these endpoints can be used. The first one to retrieve all possible ancillaries that are applicable to the leg, and the second one to add the ancillary.

Another possibility is exchange one ancillary for another one. In that case, use as well the ‘ancillary’ field as the ‘replacesAncillary’ field.

Last application is to remove an ancillary. Use only the ‘replacesAncillary’ field, and leave the ‘ancillary’ field empty (null).

Remind that ancillaries must be added in the package in the ‘definitions’ object, before referring to them in the legs.

**Parameters**: a package & leg reference and an ancillary to add or exchange, or an ancillary to replace or remove.

**Returns**: a modified package, with applied/exchanged/removed ancillary

cancel-package

**Endpoint**: POST /processes/cancel-package/execute

**Functionality**: In case of technical errors, a package can be removed. This is an optional endpoint

**Parameters**: a package reference

**Returns**: A cancelled package

GET /collections/redress-options/items   
  
POST /processes/claim-redress-options/execute   
  
POST /processes/confirm-redress-option/execute

**Endpoint**: GET /collections/redress-options/items   
 & POST /processes/claim-redress-options/execute   
 & POST /processes/confirm-redress-options/execute

**Functionality**: To provide functionality to find refund or replacement options, to claim one of them, and confirm it in a 2-phase manner.

**Parameters**: at least a package reference, and a part to be redressed: a guaranteeId, a leg reference, an offer reference, a traveller reference or ancillary reference. The returned options can be claimed, returning a modified or replacing package. After the confirmation, it will be a valid package again.

**Returns**: Refund options (from the GET), a claimed or confirmed package.

# Appendix B: OGC Meta data

By complying to the OGC API Records, Features and Processes, we have to implement a set of endpoints, facilitating ‘discovery’. Once the root of the API is published and found, the reseller should be able to discover the API.

OGC prescribes there that the discovery (the meta-data) must be available in a human and in a machine readable format.

The set of OGC endpoints that must be implemented are (in html and json):

* ‘/’ – the root or landingpage
* /api – containing the OpenAPI definition
* /conformance – gives a list of standards used in the API
* /collection – returns a list of available collections
* /collections/{collectionId} – returns a description of the collection
* /processes – returns a list of available processes
* /processes/{processId} – returns a description of the process

All these endpoints deliver a fixed output, so it is normally a one-time effort to create them.

# Appendix C: OGC structure

In the API we comply to the OGC standards, especially the OGC API Records, Features and Processes. This makes it easy to understand, and have a structure to hang on to.

### Records & Features

All requests for information are handled by a

GET /collections/{collectionId}/items

where you can filter the collection based on provided arguments.

The result (response) of a call to one of these endpoints do always have this structure (unless it is described and underpinned why it is different):

All results have the format of:  
{ “type”: “<conceptName>Collection”   
, “<conceptsName>”:   
 [ { “type”: “<conceptName>”  
 , “geometry”: *only applicable in OGC API Feature calls* , “id” : *id of the instance*  
 , “properties”: {   
 “type”: “<conceptName>”,  
 , “id” : *id of the instance*  
 … *(other properties of the concept)*…   
 }  
 , “links”: [ *…available detail links AND links for possible operations on this concept …*]  
 } *(end of concept 1)* , { *concept 2* }  
 ]  
 , “properties”: { *… properties of the collection …* }  
 }  
 , “links”: [ *… links for the collection, like next part of the data set …* ]  
}

If the endpoint is OGC API Feature compliant, the conceptName is ‘Feature’, but in the properties, the type will remain the requested concept type. This will result in a valid GeoJSON.

This structure allows us to structure things. The ‘properties’ of the feature will contain Transmodel compliant concepts and the links of the feature will expose possible next steps for the concept (link ‘purchase’ or ‘2-phase-purchase’ for an offer). The links could also contain additional information, like ‘details’, where you will be able to ‘open’ the offer, and look at the details of the legs and products.

At the bottom of the response, you’ll find the ‘properties’ of the collection, like ‘type’: ‘offers’, which will clarify that this is a list of offers. The links at the bottom will again provide additional information, or things like ‘next’, referring to the next part of the result.

### Process

All request that modifies data, have this format:

POST /processes/{processId}/execute

The only exception on this rule is ‘search-offers’, but this is due to the limitations of the http-get. According to the rules, it should have been a collection.

The format of all process requests have this format:  
{   
 “inputs”: { *the functional input arguments* }  
, “subscriber”: { this part is only relevant when you work a-synchronously, what we support as well }  
}

In the documentation below, we will only describe the ‘inputs’ part.

The output of the processes described return very often a single, but complete package.

# Appendix D: Authentication

The not open part of the API requires to obtain access and authorisation by supplying a JWT (‘token’, RFC 7519) to identify yourself, and so you can be authenticated and authorized to use the endpoint.

### Authentication flows

The several ways to get access to data sets or operations are:

* Using PKI, in the handshake protocol (mTLS) the certificate is delivered and the JWT token can be created using the credentials in the certificate when a JWT token request is executed
* Using OAuth (Client credentials flow), to obtain a JWT token
* Using username/password to obtain a JWT token
* Using a fixed token, provided in another way

To obtain the JWT token, endpoints are provided in the API, but they COULD be implemented by independent Identity Provider.

The following RFC documents apply:

* [RFC-7519](https://tools.ietf.org/html/rfc9457) which explains what a JWT token is;
* [RFC-6749 Section 3.2](https://datatracker.ietf.org/doc/html/rfc6749#section-3.2) which defines OAuth2 and the token endpoint involved in the creation of tokens;
* [RFC-6749 Section 4.4.2](https://opensource.zalando.com/restful-api-guidelines/#section-4.4.2) which defines the use of client credentials to obtain an access token;
* [RFC-7521](https://datatracker.ietf.org/doc/rfc7521/) laying the groundwork for cryptographic client assertions;
* [RFC-7523 Section 2.2](https://martinfowler.com/articles/richardsonMaturityModel.html#section-2.2) which describes how to properly secure the token endpoint with modern cryptography, thus not relying on static secrets;
* [RFC-8725](https://datatracker.ietf.org/doc/html/rfc6749) which gives guidance on securely validating and using JWTs.

### JWT structure

The JWT token itself is a standard JWT token. In the so-called ‘payload’ these fields MUST be available:

|  |  |
| --- | --- |
| iss | The issuer, like “iss”: “https://railwayundertaking.com/” |
| sub | The user id, that can be   * the (O) or (CN) from the certificate, * the client\_id from the client credential flow or * the username from the username/password authentication |
| exp | Expiry time (UNIX timestamp, in seconds) |
| iat | Issuing time (UNIX timestamp, in seconds) |

There are also a few optional fields:

|  |  |
| --- | --- |
| nbf | Not before (this token is only valid after this UNIX timestamp, in seconds) |
| jwt | A unique identifier for this JWT |

The ‘header’ part of the JWT is complying to the JWT standard, specifying explicitly that the used encryption method is SHA 256 and the type is JWT (and not JWS or JWE):

{ "alg": "HS256", "typ": "JWT" }

### Authentication endpoint

The authentication endpoint, that will deliver a JWT as described above, can be found in the ‘tech’ module of the specification.

At the end, the logical flow for all not-open endpoints is to request a JWT using the token endpoint, and supply in the header fields of the GET or POST requests. The JWT token can be supplied using a username/password, a OAuth2 client credential flow or a mTLS request, where the last one depends on a shared ‘base’ certificate authority, allowing usage in setups without knowing each other in advance.

POST /oauth/token { username/password }

POST /oauth/token { client\_id/client\_secret }

POST /oauth/token () { PKI (mTLS) }

# Appendix E: Authorisation

Within the API we can distinguish open data sets, data sets that could be open and data sets that require authorisation.

### Open endpoints

The open endpoints can be found in the API description, they are solely marked with

security:  
 - None

|  |  |
| --- | --- |
| / | **Discovery:** Landing page, in HTML and JSON, providing general information and ‘bootstrap’ links. |
| /conformance | **Discovery:** Exposing all formal formats, data standards etc that are used. |
| /api | **Discovery:** A formal OpenAPI specification of the API |
| /collections | **Discovery:** Describing all collections that are available in the API (open and secured) |
| /processes | **Discovery:** Describing all processes facilitated in the API (normally secured) |
| /collection/{collectionId} | **Discovery:** Describing the information that is in that specific collection, including the format |
| /processes/{processId} | **Discovery:** Describing the information that is in that specific collection, including the format |

### Secured endpoints

The majority of the endpoints needs authentication & authorization, they are marked with other security schemes, like BearerAuth (authenticatie with a JWT, obtained anywhere), OAuth (also a JWT authentication, but the token is obtained by the client identity flow) or OAuthPKI (this requires mTLS, the token is obtained by supplying a PKI certificate).

|  |  |
| --- | --- |
| /collection/{collectionId}/items | Containing the data of collections like refund-options, assets, ancillaries, travel-documents |
| /processes/{processId}/execute | Triggering a process on packages, legs, products, etc. You need to be authenticated & authorised to execute these. |

# Appendix F: Pagination

OGC API Record and Features describe a standard way to handle pagination for large data sets. Roughly said, there is at the end of a result (a collection) a facility for ‘links’, that include links to the next (and previous) part of the complete data set. That requires also that there are a few arguments by default:

* Offset (where the requested part of the data set should start)
* Limit (the number of rows/features that have to be returned at max)

*Example*

* First, we call /collections/{some collection}/items, and it returns the first 100 entries.
* At the bottom of the result, there is a link, using the relation “next” and the url specified there is collections/{some collection}/items?offset=100&limit=100, to retrieve the next 100 entries.
* Using the link will return up to the next 100 entries, and at the bottom, we’ll find 2 links: next - collections/{some collection}/items?offset=200&limit=100 and previous - collections/{some collection}/items?offset=0&limit=100

# Copyrights

Licensed under the Apache License, Version 2.0 (the "License");  
you may not use this file except in compliance with the License.  
You may obtain a copy of the License at  
  
 http://www.apache.org/licenses/LICENSE-2.0  
  
Unless required by applicable law or agreed to in writing, software  
distributed under the License is distributed on an "AS IS" BASIS,  
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
See the License for the specific language governing permissions and  
limitations under the License.