Eclipse Integration Guide

V 1.18

04/10/2021

Ukheshe Technologies

Design Quarter, William Nicol Avenue, Fourways

Johannesburg

Table Of Contents

Table Of Contents 0

Introduction 7

About Eclipse 8

The Eclipse Architecture 10

Naming & Conventions 14

Environments & Endpoints 19

General Guidelines 22

Encryption 24

Authentication & Authorisation 26

- Simple Identity/Password 26
- Identity/Password + TOTP 27
- Identity/Password + PKI (And potentially TOTP as well) 28
  - Prerequisites 28
  - Authentication Flow 29
  - Java Code Example 30

High Level Entities in Eclipse 36

Wallet Hierarchy & Fund Flows 38

Use Cases 40

- Initial Prerequisite - Tenant onboarding 41
- Create a Tenant 41
- Set up Admin Users within the Tenant 41
  - Prerequisites 41
  - Step 1 - Create profile for tenant admin user 41
- Create an Initial Wallet Type 41
  - Prerequisites 44
  - Step 1 – Create wallet type 44
- Change Tenant System User Username and Password 44
  - Prerequisites 44
  - Step 1 – Request a new tenant username and password 44

Standalone Customer Use Cases 46
Register a new customer with a digital wallet 46
Prerequisites 46
Step 1 – Create Customer 46
Step 3 – Add KYC Doc info 46
Step 4 – Kick off Ratify 47
Step 5 – Create Wallet 47
Register a new customer and issue a virtual card 47
Note that as prepaid card wallets always have an initial card it will be created when the Wallet is created. 47
Prerequisites 48
Step 1 – Create Customer 48
Step 3 – Add KYC Doc info 48
Step 4 – Kick off Ratify 48
Step 5 – Create Wallet 48
Register a new customer and issue a physical card 49
Step 4 – Create Wallet 49
Issue a secondary physical card 50
Prerequisites 50
Step 1 – Create Card 50
Give Customer Eclipse API Access 50
Prerequisites 51
Step 1 – Create Customer Identity and Password 51
Search for a customer 51
Prerequisites 51
Step 1 – Search customer 51
Physical Card PIN Set 51
Prerequisites 51
Step 1 – Set PIN 52
Physical Card PIN Counter Reset 52
To reset a Card PIN counter if it has previously been exceeded. 52
Prerequisites 52
Step 1 – Reset PIN 52
Physical Card PIN Change 52
Prerequisites 53
Step 1 – Reset PIN 53

Transfer Funds Between Wallets 53
Prerequisites 53
Step 1 – Wallet to Wallet Transfer 53
Transfer from Eclipse SoV to Non-Eclipse Sov 54
Transfer from Non-Eclipse SoV to Eclipse Sov 54
Cross Tenant Transfers 55
Get Wallet Balance 56
Prerequisites 56
Step 1 – Wallet request 57
Get Wallet Transaction History (Statement) 57
Prerequisites 57
Step 1 – Transactions request 57
Stop a Card 57
Prerequisites 57
Step 1 – Stop Card 57
Unstop a Card 58
Prerequisites 58
Step 1 – UnStop Card 58
Prerequisites 58
Step 1 – Enable / Disable Ecommerce on Card 58
Prerequisites 59
Step 1 – Enable / Disable International 59
Do a Physical Card Replacement 59
Prerequisites 59
Step 1 – Create Card 59
Do a Virtual Card Replacement 60
Prerequisites 60
Step 1 – Create Card 60
Retrieve Card Details for Customer (API) 60
Prerequisites 60
Step 1 – Create a CAP (Card Access Password) 60
Step 2 - User enters CAP, then Retrieve Card 61
Retrieve Card Details for Customer (Captive Portal Flow) 61
Step 1 – Create Card 61
Prerequisites 61
Step 2 – Send user to webview to set password for Card 62

Prerequisites 62
Step 3 – View Card 62
Prerequisites 62
Prerequisites 63
Step 1 – Update Customer information 63
Prerequisites 63
Step 1 – Update the customer address 63
Prerequisites 63
Step 2 – Update the document 64
Cancel Customers Wallets & Cards 64
Prerequisites 64
Step 1 – Delete the wallet 64
Prerequisites 64
Step 1 – Update profile, addresses, documents 64
Step 2 – Validate Customer using Ratify 64
Prerequisites 65
Step 1 – Create QR Code linked to the wallet 65
Prerequisites 65
Step 1 – Get QR Code details 65
Wallet Top-Up 65
Prerequisites 65
Step 1 – Initiate topup process 66
Alternatively EFT into Pool account 66
Wallet Withdrawals - Non Card 66
Prerequisites 66
Step 1 – Initiate withdrawal process 66
Create a Card to Card Money Send request (Not available yet) 67
Organisation Use Cases 68
Create an Organisation 68
To create an organisation 68
Prerequisites 68
Step 1 – Create Organization 68
Step 3 – Add Supporting Documentation 68
Step 4 – Kick off Ratify 68
Step 5 – Create Wallet 69
Add Positions to Organisation 69

To link a customer as having a position in an organisation 69
Prerequisites 69
Step 1 – Create Customer Position within the Organisation 69
Register a new customer as part of an organisation and issue a virtual card linked to an Organisation wallet 69
(With the creation of the Wallet, the card is automatically created) 70
Step 4 – Create Wallet under Organisation 70
Prerequisites 71
Step 1 – Update profile, addresses, documents 71
Step 2 – Validate Organisation using Ratify 71
Create an Organisation Wallet 71
To create an Organisation Wallet. 71
Prerequisites 71
Step 1 – Create an organisation wallet 71
Request an Organisation Wallet Balance 72
Prerequisites 72
Step 1 – Request organisation wallet balance 72
Get Wallet Transaction History (Statement) 72
Update Organisation information 72
Prerequisites 72
Step 1 – Update organization Information 72
Update Organisation address 73
Prerequisites 73
Step 1 – Update organization address 73
Update Organisation documentation 73
Prerequisites 73
Step 1 – Update organization documentation 73

Payments Use Cases 74
Payment Type: ZA_QRCODE 76
Payment Type: GLOBAL_PAYMENT_LINK 78
Payment Type: GLOBAL_VAS 79
Payment Reversals 79
Payment Status State Diagram 80

Global Integrator Use Cases 82
Authentication 82

Customer/Wallet Lookup 82
Wallet Transfer to Customer 84
Direct Withdrawal 85
Integrator Wallet Enquiry 88
Transfer Transaction Query 90
Reconciliation Query 90

Exposing A Tenants Store of Value as a wallet type in Eclipse 93

Tenant Config Reference 96

Reporting 100
Document Database 100
Ad Hoc Reports API 101

Postman 103
Why use Postman 103
How to use 104

Application Error Codes 107

FAQ 108

Sandbox Testing Accounts 111
Wallet Withdrawals 111
Wallet Topups 112

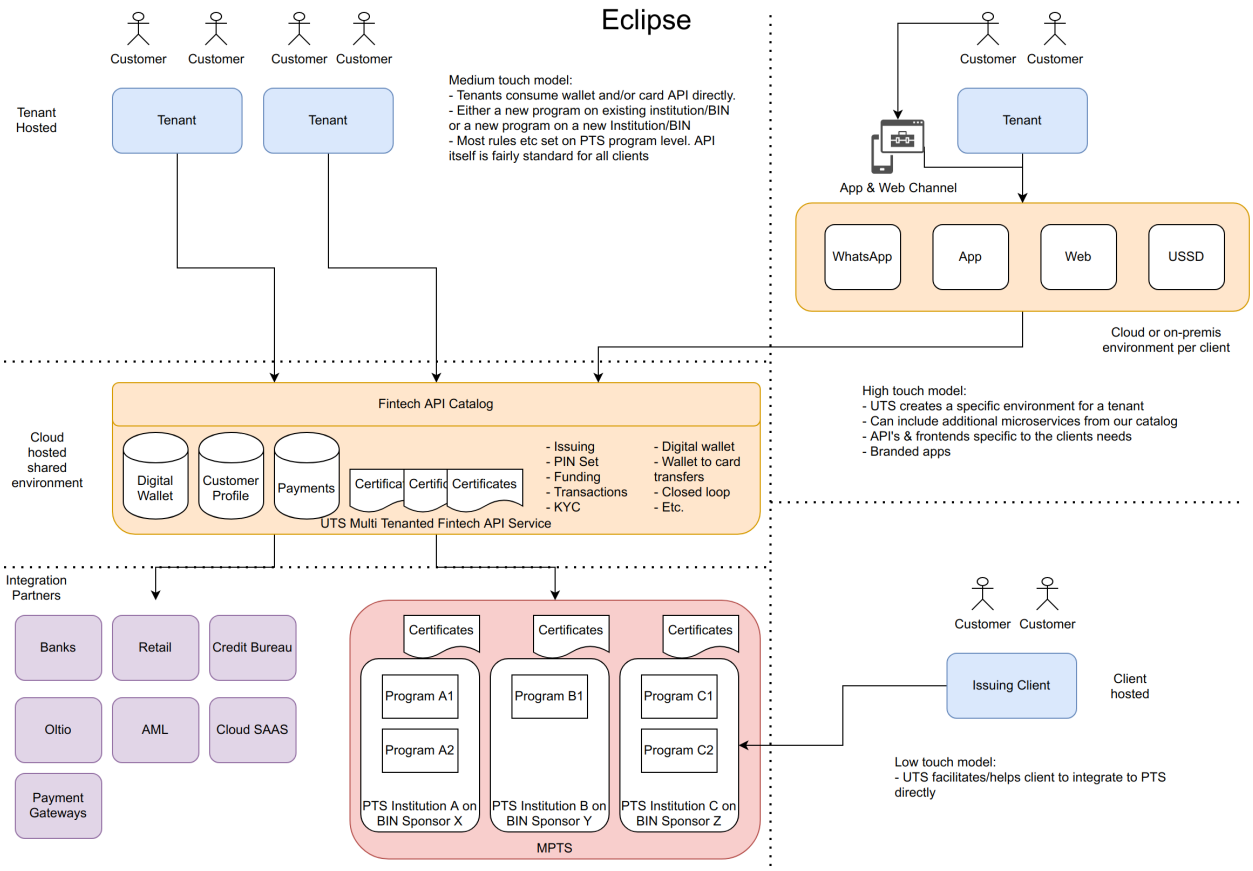
Appendix 1: Production Setup Prerequisites 113
Functionality Dependent Prerequisites 113
Technical Prerequisites 115

Introduction

This document is intended as a guide for any third party product owners or developers intending to integrate with Eclipse as an open, feature rich digital wallet and card issuing platform. Eclipse has an extensive catalog of API's available for tenants to manage the lifecycle of an Eclipse Customer from onboarding through to wallet creation, card issuing, topping up, wallet transfers, withdrawals and so forth.

The document refers to the Eclipse Sandbox environment for all examples and anyone can access this environment and start to code against the Eclipse API's for POC's, demonstrations and testing.

About Eclipse



Eclipse is a multi-tenanted cloud-based API and associated admin console for enabling banks, fintechs and telecom operators to build flexible and compelling financial services propositions across the globe. Eclipse is in now way associated with the Eclipse foundation nor the Eclipse IDE. The naming is a pure coincidence.

Eclipse is built and operated by Ukheshe Technologies based on hundreds of man-years of experience in the financial services, telecoms and payments arena. The goal of the platform is to provide a set of powerful API's covering everything from customer & organisation management to digital wallets, payment gateways, card issuing, card acceptance, bank integration, authentication, KYC, QRCode acceptance/payments and numerous other capabilities required by most Fintech propositions.

The platform allows one to focus on product differentiation and time to market while Eclipse takes care of the heavy lifting, partner integrations and ensuring a bullet-proof, scalable, cloud-based platform to leverage in one's own offerings. While normally rolling

out a proposition involving digital wallets, card issuing and acceptance, payment gateways etc. requires a myriad of compliance and technical hurdles. Eclipse takes care of this and separates the engine room of payments and card management from the end-user channels and propositions. Whether it’s a rewards platform, online-auction App, corporate card issuing, WhatsApp based digital wallet or anything in between, Eclipse provides the building blocks necessary to delight one's customers.

While Eclipse can be used as a pure API-first platform, Ukheshe technologies can also provide professional services and custom software development to help Fintechs build and operate propositions on top of Eclipse. There are 3 key approaches to engaging with Eclipse/Ukheshe:

1) A high-touch model where Ukheshe can use the Eclipse API’s and other technologies to build customer facing propositions for the tenant.
2) A medium-touch model where a tenant builds their own solutions using the Eclipse platform.
3) A low touch model where Ukheshe technologies acts as a consultant/partner in aiding an organisation to build their own integrations into some of the platforms underlying Eclipse (E.g. MPTS).
4) An global integrator model where the integrator is not a tenant on the platform but can disburse value from a global wallet of theirs to wallets on any participating tenant.

The Eclipse Architecture

Eclipse is an API-first SAAS hosted on AWS by Ukheshe. At the simplest level, it's a set of 150+ restful endpoints all carefully designed to provide a consistent, powerful yet simple set of services covering everything from customer onboarding, to wallets, card issuing and payments. Eclipse has an admin portal that consumes the API's for adhoc, CRM and general administration tasks. The intent however is that most tenants of Eclipse would utilise their own branded customer facing channels, and those would in turn call Eclipse directly or via their own backend servers. Accompanying the admin portal and API's are a set of whitelabel starting points for tenants interested in building channels on top of Eclipse (or having Ukheshe build channels for them):

- A whitelabel WhatsApp bot covering all the core functionality of onboarding, wallets, card issuing, QR payments and VAS
- A whitelabel React based PWA covering all the core functionality of onboarding, wallets, card issuing, QR payments and VAS
- A whitelabel React native based App covering all the core functionality of onboarding, wallets, card issuing, QR payments and VAS

The combination of the whitelabel channels and Eclipse API allows tenants to go to market quickly and evolve their proposition.

The Restful API's exposed by Eclipse all point to the same URL for all tenants. In the URL's there is a path parameter that determines the tenant calling the API and this is also enforced by the JWT bearer token passed on each API request. As an example, tenant 100 could get a list of their customers with a GET request to:

https://ukheshe.live/eclipse-conductor/rest/v1/tenants/100/customers

Whereas tenant 200 would use:

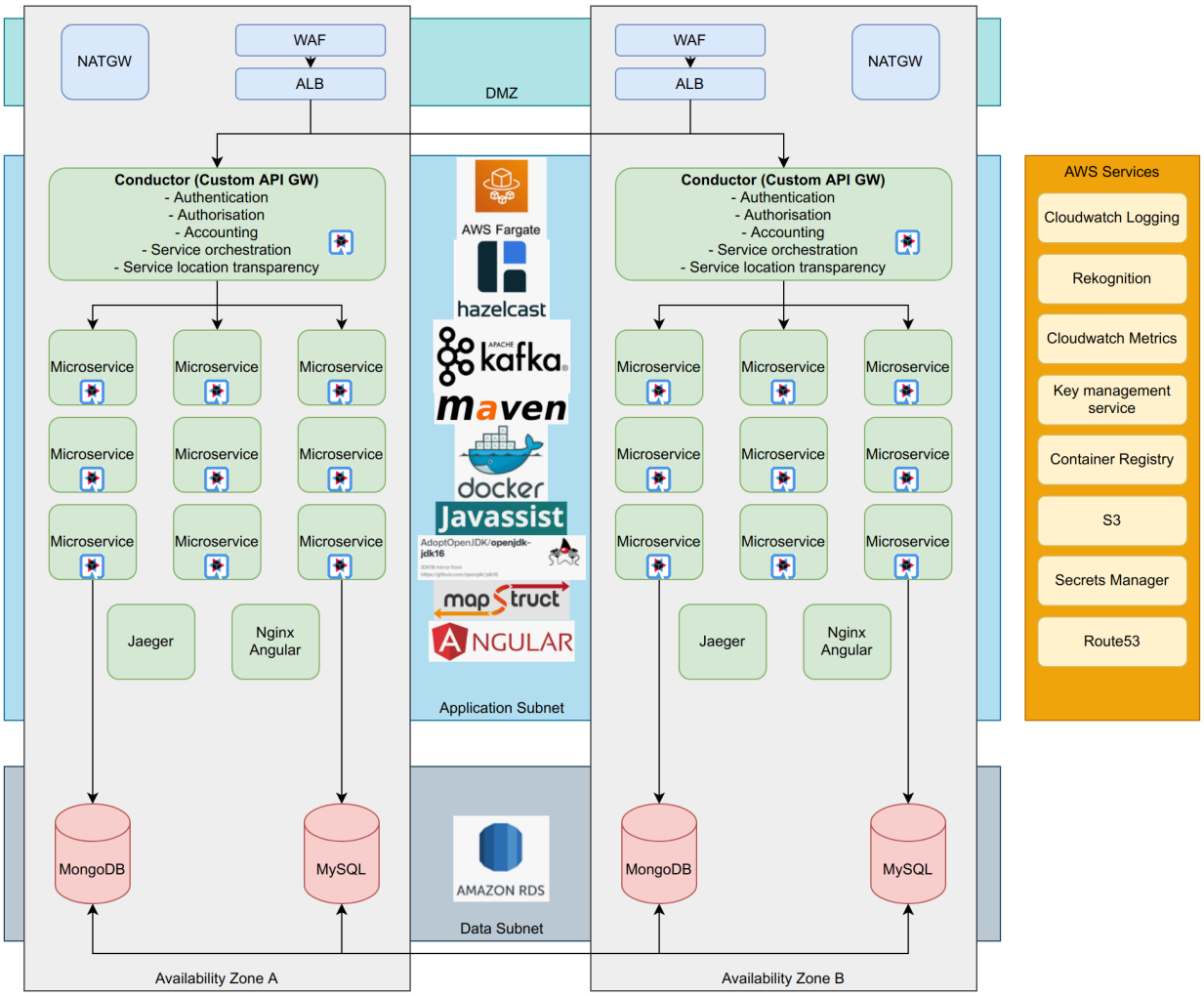
https://ukheshe.live/eclipse-conductor/rest/v1/tenants/200/customers

Tenants are logically separated on the backend. Each has its own configuration, own customers, own wallets etc but shares the underlying AWS compute and persistence services. This is similar to Office365 or Google Apps - an organisation has their own settings and environment but all hosted across a farm of underlying shared infrastructure.

The Eclipse backend is layered, utilising best of breed components in the various layers to provide a modern scalable cloud native architecture. These layers are built up as follows:

1. AWS datacenter infrastructure (compute, storage and networking)
2. AWS Elastic container services for container based deployments, autoscaling and resilience
3. AWS Relational database services and document database services for data persistence
4. Docker based containers using Linux as a base
5. Java 17 runtime
6. Quarkus framework
7. Numerous open source libraries
8. Ukheshe standardised architecture libraries
9. Eclipse business logic code

The high level architecture is depicted in the following diagram:



Although Eclipse as a whole serves as an orchestration layer providing coarse-grained APIs hiding the complexity of multiple complex fine-grained APIs, it is a lot more than simply an ESB or API gateway. Many of the core constructs in Eclipse are implemented as microservices and have state and persistence (e.g. customers, wallets etc). The API's and microservices are designed in such a way that implementations can be externalised or hosted internally in Eclipse. E.g. a wallet's Store of value could sit in Eclipse using its wallet functionality, or point to an external store of value such as a bank account, crypto platform or card management system. The Eclipse API remains consistent and yet the implementation behind it can evolve or change. A good example is card issuing - the API calls on Eclipse to issue a card to a customer are very simple - as few as one or 2 fields in a POST of JSON and a card is issued. The backend card management system could be Mastercard PTS or a Postilion postbridge interface or anything in between - and yet none of this is the concern of the API caller.

One of the core technology enablers in Eclipse is the component known as the eclipse-conductor. This is the entry point for all API calls. The conductor covers many of the functional blocks seen in a combination of an API gateway and ESB and was built by Ukheshe to satisfy the use cases of Eclipse and overcomes many of the issues the team has faced using off the shelf API gateways and ESB's that require numerous components and typically awkward composition logic languages such as BPEL or Lua scripting. The conductor is built on top of Vert.X/Quarkus using a completely stateless non-blocking IO model for huge scalability and performance and allows for orchestration logic to sit in Java. The core functional capabilities are as follows:

- Identity provider / Authentication server supporting JWT, OIDC, SAML2 and custom auth schemes
- API tracing
- API metrics (latency, error rates, concurrency etc)
- Load balancing & failover
- Java based service composition logic
- Reverse proxy
- Service discovery
- Data validation
- Caching
- Authorisation logic & policy enforcement
- Web application firewall (XSS, SQL injection)

- Accounting
- Rate limiting (throughput and concurrency)
- TLS termination

The Eclipse conductor either proxies to or orchestrates across downstream microservices which again are largely built using Java/Quarkus.

Naming & Conventions

In order to best understand the API's and their functionality, its key to cover the various names and conventions used in this document along with the Swagger

Name	Description
Authentication	The process whereby a customer provides an identity & password to prove they are who they say they are and in response are given a token (JWT) to use on any further API calls showing who the call is being done by
Admin User	A user with an identity which can administer a tenant and its customers and organisations. Tenants would have admin users for their employees and API access.
Identity	A tenant or customers username used in conjunction with a password and possibly one time pin or public key when authenticating
JWT	JSON Web Token used as an Authorization header on all HTTP calls to indicate to the backend which authenticated tenant/customer is doing the call
wallet	A monetary balance and associated transaction history. The underlying store of value depends on the type of wallet. It could be a wallet for a card or a pure digital wallet.
Customer	A customer of the tenant. A physical human being with name, details etc.
3DS	3-D secure - A process whereby an issuing bank interacts with their cardholder via an iFrame or similar channel to verify that the person doing the transaction is the cardholder. Typically via a OTP or banking app auth process.
Transfer	A funds movement consisting of a debit on one wallet and a credit on another. Both the sender and recipient end up with a transaction on their transaction history. Transfers always happen between two Eclipse wallets.
Username	Same as identity. Used for logging in
customerId	A unique system generated numeric identifier for a

	customer which never changes for the lifetime of the customer. E.g. 12345
organisationId	A unique system generated numeric identifier for an organisation. Typically only used for large partners integrating with Eclipse who want to sign up customers and indicate that those customers fall under a certain structure/organisation.
idNumber	The legal reference number for a customer - e.g. a national identity number or passport number.
availableBalance	The amount of money in a customer's wallet which is available to be spent, transferred or withdrawn.
currentBalance	availableBalance + reservations. The money that is earmarked as belonging to the customer but not all of it may be available to them as some may be locked (reserved) for current transactions in progress.
reservations	Funds in a wallet which cannot be spent as they are set aside for a transaction which is in progress or have not cleared yet. Reservations sum up to the difference between current balance and available balance.
phoneOtp	A 6 digit code SMS'd to a customer (OTP) as a means of verifying the customer has access to their registered phone number.
deposit	A process whereby a customer can give cash to an Eclipse retail partner (e.g. Pick 'n Pay) and have the corresponding amount show in their wallet. This is equivalent to a cash topup.
ratify	Eclipse's platform for verifying the KYC details of a customer or Organisation. This is the process of capturing a customer's identity document, selfie and details and verifying they all match using OCR and facial pattern matching.
Transaction	An entry on the Eclipse ledger representing a debit or credit on a wallet.
Transient QR Code	A temporary QR Code representing a fixed payment amount to be credited to the associated wallet if scanned & paid.

	These are typically Masterpass QR Codes and are a 10 digit number that can end up being recycled and reused and have an expiry date. They are used if a customer wants a QR Code that can be scanned and automatically show a fixed amount to be paid.
Withdrawal	The opposite of a deposit. A withdrawal is initiated by a customer and allows them to receive cash at a partner retail store and the corresponding amount is debited from the customer's wallet.
topup	The process of a customer using a bank account, credit card, debit card etc to increase the balance of their Eclipse wallet.
baseURL	Eclipse sandbox environment: https://eclipse-java-sandbox.ukheshe.rocks/eclipse-conductor/
externalUniqueId	API's which result in a transaction on the Eclipse backend include a unique string passed by the caller which can be used to ensure that duplicate transactions are not processed more than once if there is some network issue or accidental transaction replay. These should be passed as a GUID/UUID created by the API caller or some other concatenated string uniquely identifying the request. These must be unique across all tenants and max length 50 characters.
externalId	Some APIs allow the caller to pass in their own transaction identifier for a transaction which can be used for reconciliation purposes. One can use the uniqueId for this as well but sometimes callers want to use the same externalId for multiple transactions which is not allowed with uniqueId but is allowed with externalId.
TenantId	Unique identifier for a tenant. Created by the system when a tenant is created

Wallet Type ID	Unique identifier for a type of wallet
Organisation	Organisation is typically a business or group of people to which the customers are associated with. A tenant may add multiple organisations. (Any international transactions must be either reported against either an Organisation or Customer, not a Tenant)
Address	The address of the customer or organisation.
Position	The defined customers position within an organisation that may allow Ukheshe to contact them for specific queries (such as support or accounts) and may be used to define specific access rights to that organisation.
System Identity	The identity of a tenant system used to authenticate with Eclipse
Card	A physical to virtual payment card, such as an EMV Mastercard.
PKI	A public key infrastructure (PKI) is a set of roles, policies, hardware, software and procedures needed to create, manage, distribute, use, store and revoke digital certificates and manage public-key encryption. The purpose of a PKI is to facilitate the secure electronic transfer of information for a range of network activities such as e-commerce, internet banking and confidential email. It is required for activities where simple passwords are an inadequate authentication method and more rigorous proof is required to confirm the identity of the parties involved in the communication and to validate the information being transferred.
walletFriendlyId	Wallets have two unique identifiers - one is a system generated numeric (walletId) and the other is a system

	generated string (walletFriendlyId) of length 8 which is a mixture of CAPS alphanumeric and numeric which does not contain characters such as O,0,L,I as these can be difficult to distinguish in some fonts. An example walletFriendlyId: R982DFP2. 2 walletFriendlyId's are unlikely to be similar and hence typing them and getting some characters wrong is unlikely to match a different wallet.
Store of Value (SoV)	The actual end platform that has the master of a wallet balance and ledger. For some wallet types, Eclipse is the store of value and for others such a prepaid cards, the SoV could be an issuing platform like PTS.

Environments & Endpoints

Eclipse has a production and sandbox environment which are completely separated. Tenants are encouraged to develop and test against the Sandbox environment which is fully functioning other than the ability to topup and withdraw with real money. Each environment has its own API endpoint:

Eclipse live environment:

https://ukheshe.live/eclipse-conductor/

Eclipse sandbox environment:

https://eclipse-java-sandbox.ukheshe.rocks/eclipse-conductor/

The Swagger is available at
https://eclipse-java-sandbox.ukheshe.rocks/eclipse-conductor/docs/#/

See Postman section for a collection of predefined requests.

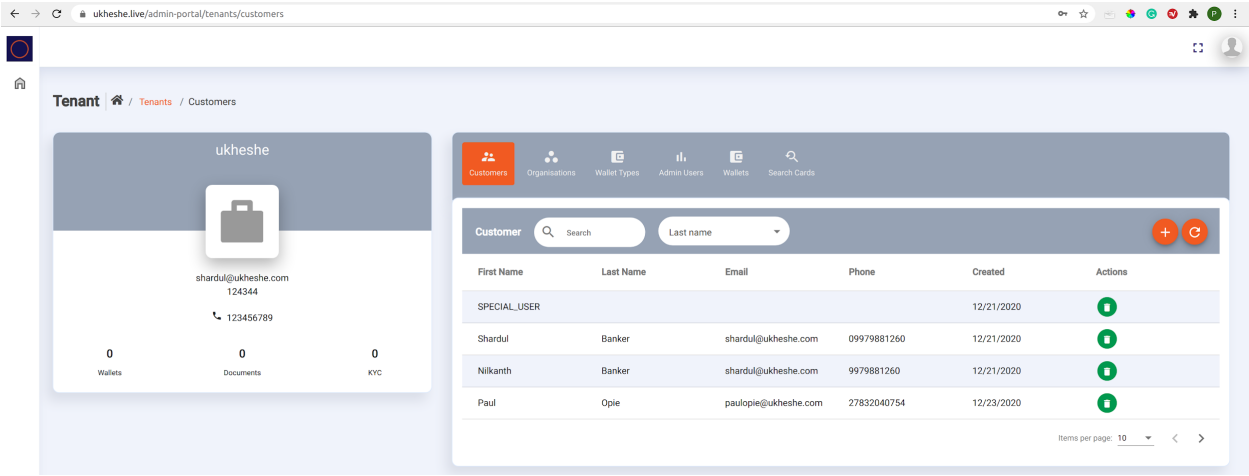
Each of the above URL's is referred to as {baseURL} in the documentation and serves as a placeholder for the first part of a URL. E.g. {baseURL}/rest/v1/aaa on Sandbox would be https://eclipse-java-sandbox.ukheshe.rocks/eclipse-conductor/rest/v1/aaa

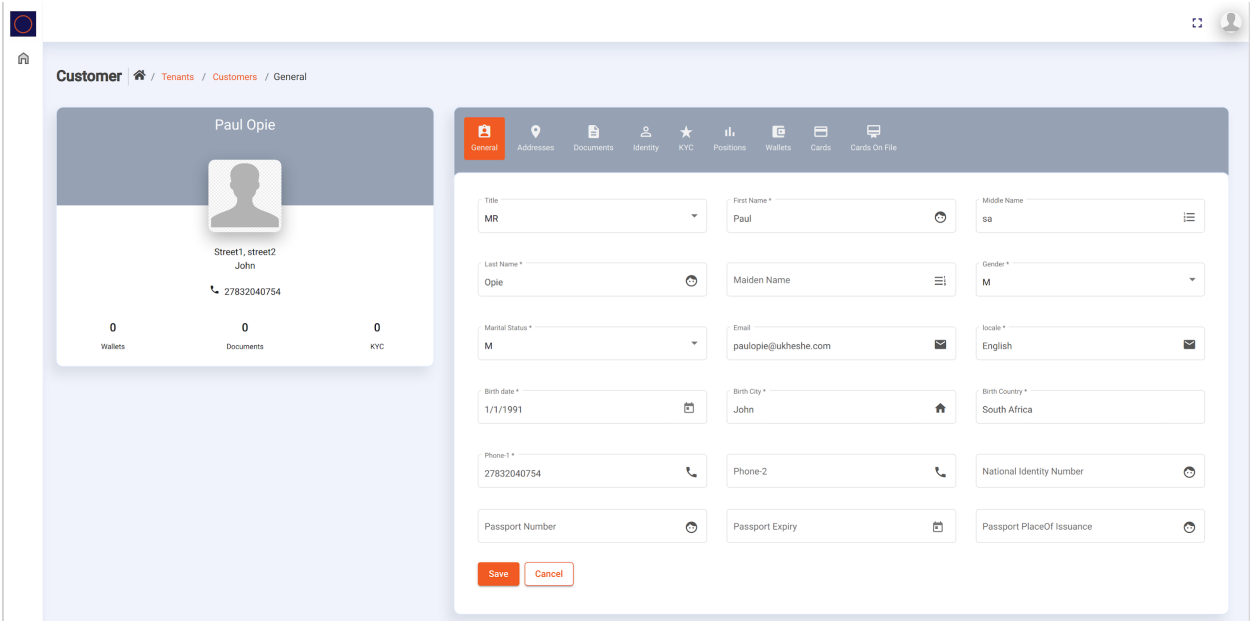
Admin Portal:

- Production: https://ukheshe.live/admin-portal/
- Sandbox: https://eclipse-portal-admin-sandbox.ukheshe.rocks/admin-portal/

Tenants can request sandbox credentials in order to try it out.

The admin portal uses the Eclipse API's so it can be a great way to look at the API's in use and the payloads that are sent/received.





In order to test QRCode payments on Sandbox, one can use any Masterpass App (e.g. Vodapay App) and put it into test mode by scanning the following QRCode:

91109
10198
40075

Once in test mode, enroll card 500200010001052234 with any PIN, CVV and Expiry and use it to pay Eclipse QRCodes.

General Guidelines

1. The API's are ReSTful and available over TLS using HTTP/1.0, HTTP/1.1 and HTTP/2
2. Message bodies are JSON
3. API's are physically hosted in Ireland so network latency from Africa is about 180ms
4. Other than some of the ratify API's that can take a few seconds, most API's have a latency of <100ms (excluding network latency)
5. All API requests are audited and tracked to the user and IP address. Attempts to circumvent the system security, send inappropriate load, or perform fraudulent activities will be tracked down and dealt with to the full extent of the law. If API users discover vulnerabilities or are able to exploit the system then please report as such to info@ukheshe.com.
6. Most API's that return a list of results can be passed query parameters offset and limit. The offset is a 0 based index of which result to return from. Defaults to 0. The limit is the number of results to return and typically defaults to 1000. These can be used for pagination when required but most often the defaults are fine.
7. APIs use HTTP response codes as follows:
    a. 404 : The resource could not be found. E.g. a request for a specific customer, wallet etc was not found in the database.
    b. 400 : Bad input data in a query parameter or body found. E.g. a parameter was invalid or incorrect in the context of the request.
    c. 401: Bad authentication credentials or JWT
    d. 500 : Something unexpected went wrong processing the request. This would typically be a system level error and is likely to work if re-tried later.
    e. 204 : With no response body means the resource was created successfully or the operation completed successfully.
    f. 200 : Success with a response.
8. In addition to the HTTP error code, errors return with a body with a list of errors (normally only one). Refer to the section on application error codes for details E.g.:

```
[
  {
    "code": "USR007",
    "description": "Incorrect code",
    "severity": "LOW",
    "type": "BUSINESS",
    “traceId”: “hchuewhuiehdiwhei”
```

```
  }
]
```

9. The error type can be BUSINESS or SYSTEM. SYSTEM indicates a system error and may work if retried later. BUSINESS indicates a validation error or user error. Errors will normally include a traceId which can be quoted if reporting an error so that it can easily be traced by Eclipse operations support.
10. All API's perform initial data validation against the published schema on Swagger. Validation failures are reported as BUSINESS errors with error code VAL001. E.g.:

```
[
  {
    "type": "BUSINESS",
    "severity": "LOW",
    "description": "Value '' for NewEclipseTenant's companyNumber is invalid: It must be a minimum of 5 and maximum of 20",
    "code": "VAL001"
  },
  {
    "type": "BUSINESS",
    "severity": "LOW",
    "description": "Value '' for NewEclipseTenant's apiKey is invalid: It must be a minimum of 36 and maximum of 36",
    "code": "VAL001"
  },
  {
    "type": "BUSINESS",
    "severity": "LOW",
    "description": "Value 'string' for NewEclipseTenant's phone1 is invalid: It must be a minimum of 10 and maximum of 20",
    "code": "VAL001"
  }
]
```

11. The swagger contains detailed schema descriptions and field requirements. Click on the schema tab to see these:

Request body

application/json

Example Value | **Schema**

```
NewEclipseWalletType ⌄ {
  description:
                The high level mode of the wallet type

  configuration
                ⌄ [
                Weakly typed list of configuration settings on the wallet. The available settings and their impact on wallet behaviour are provided independently of the swagger API document

                Avp ⌄ {
                  description:
                                Currently supported withdrawal types.

                  att*          string
                                pattern: \S
                                nullable: false
                  val*          string
                                nullable: false
                }]
  currency      string
                maxLength: 3
                minLength: 3

                ISO 4217 currency code for the balances that will be held in this wallet

  mode*         string
                nullable: false

                The top level mode of this wallet

                Enum:
                ⌄ [ CLOSED_LOOP_DIGITAL, OPEN_LOOP_DIGITAL, PREPAID_CARD ]
}
```

Encryption

All communication with Eclipse is over a TLS 1.2+ encrypted socket with Eclipse having a valid & verified certificate with a full certificate chain leading to the publicly available Amazon root CA. Clients must not trust unverified certificates as doing so could enable a man in the middle attack.

When Eclipse does webhooks back to tenants' systems, the tenant can verify the message integrity by using a special “Eclipse-Signature” http header sent in the webhook. The signature is of the format:

Header Name: Eclipse-Signature

Header Value: t=xxx,v1=yyy

Where: xxx is a timestamp being the milliseconds since the UTC EPOCH at which Eclipse sent the webhook. yyy is a Base64 encoded HMAC-SHA256 of the timestamp concatenated with a “.” and then concatenated with the message body (If there is no message body then treat the body as an empty string). The secret key used in the HMAC is a configuration on the tenant called HMACOutboundSignatureKey.

As an example:

HMACOutboundSignatureKey=HelloWorld

MessageBody={"alias":"bluXSSCPCZ","cardOnFileId":"8e2c86a5-1acf-48b7-aa70-11a1b18c0d84","last4Digits":"0028"}

Eclipse-Signature header value:
t=1625602409535,v1=**b2+TTx2JbaHqeM67fpRye30Go1vdyvyGdoljBqdL9Rg=**

HMACSHA256 of "1625602409535.{"alias":"bluXSSCPCZ","cardOnFileId":"8e2c86a5-1acf-48b7-aa70-11a1b18c0d84","last4Digits":"0028"}" and key "HelloWorld" is:

Hex: 6f6f934f1d896da1ea78cebb7e94727b7d06a35bddcafc8676896306a74bf518

Base64: **b2+TTx2JbaHqeM67fpRye30Go1vdyvyGdoljBqdL9Rg=**

Hence signature matches

Tenants are requested to verify signatures in order to verify the authenticity of Eclipse webhooks.

For inbound message integrity verification, if the tenant sets a configuration value for HMACInboundSignatureKey and HMACInboundSignaturePositions then Eclipse will expect all HTTP requests from any users whose position is included in HMACInboundSignaturePositions to include an Eclipse-Signature header generated in the exact same way. If no value is set in the tenant configuration for HMACInboundSignatureKey then no integrity is checked. Ukheshe encourages tenants to make use of inbound integrity checks for TENANT_SYSTEM admin users by setting HMACInboundSignatureKey with a complex secret and HMACInboundSignaturePositions=TENANT_SYSTEM.

Eclipse will reject messages where the timestamp in the signature is more than 5 minutes old. Tenants can do the same. This ensures replay of old messages will be rejected. Tenants should ensure their servers utilise NTP or similar to ensure clocks do not drift.

Eclipse rejects invalid or missing signatures with an HTTP 403 and body E.g.

[{"type":"BUSINESS","severity":"INFO","description":"Missing Eclipse-Signature header to ensure message integrity","code":"SEC001","traceId":"4a02a06f5e052b14"}]

Authentication & Authorisation

All calls to the Eclipse ReST API must pass an HTTP header “Authorization” where the value consists of “Bearer <JWT>” where <JWT> is a JSON Web Token.

E.g.

authorization: Bearer
eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJiY2x1YXBAZ21haWwuYr9tIiwidWlkIjoyMDMsImNoIjoiRUt6Y0YwQnBzSyIsInJvbGVzIjoiY3VzdG9tZXIiLCJpc3MiOiJodHRwOlwvXC9qaW5pLmd1cnUiLCJleHAiOjE1NDc3Njg2MjF9.KCx9TY-yLOPkbCVrFz_sfNAGC0FVAcTksaIFs2TToaM

The Token allows the Ukheshe servers to identify and authorise the API caller on each request without needing to pass the password on each API call. It tells the server who the caller is and the permissions they have. To obtain a token, there are 2 options - one with simple identity/password and one more secure which in addition, uses PKI to allow the client and server to guarantee the authenticity of each other. Eclipse does not enforce the use of PKI as simple tenants may choose not to use this, but the risk sits entirely with the tenant as without PKI, a compromised tenant password could allow anyone to impersonate the tenant, and unlike private keys, passwords are transmitted over the wire on auth requests making them more susceptible to exposure.

Simple Identity/Password

HTTP Method: POST

URL: {baseURL}/rest/v1/authentication/login

Content Type: application/json

Body:

```
{
  "identity": "0800000000",
  "password": "sandbox"
}
```

The identity is the customers/admin users username and the password is their password.

Assuming the identity and password are valid, the API will return a body such as:


```
{
   "expires" : "2020-07-24T09:49:58Z",
   "expiresEpochSecs" : 1595584198,
   "headerName" : "Authorization",
   "headerValue" : "Bearer
eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiIwODAwMDAwMDAwIiwidWlkIjo1NjQsImNoIjoiNWlQeFo2N1Y3NiIsInJvbGVzIjoi
Y3VzdG9tZXIiLCJzZXNzIjoiOWRhMzNiYzktYzhiZS00NGNlLTk3M2MtZGZjNDVkZTBjZDc3IiwiaXNzIjoiaHR0cDpcL1wva
mluaS5ndXJ1IiwiZXhwIjoxNTk1NTg0MTk4fQ.kAGOMTQFZ8v215zHJ9syrNrE7SOdoS39iGwA_WkN-fU",
   "roles" : [
      "customer"
   ],
   "sessionId" : "9da33bc9-c8be-44ce-973c-dfc45de0cd77"
}
```


The expires value indicates when the token expires and a new one should be requested as the current one will no longer work. This is also provided in a code friendly expiresEpochSecs value which is the number of seconds after 1970/01/01 UTC. The headerName tells the caller what to name the header (it's not case sensitive) and headerValue is what the headers value should be.

Tokens expire after 15min but this may change and hence the expiry should be checked and adhered to. Expired tokens can be renewed within 30 days of expiry by posting the JWT to the /renew endpoint. A new fresh JWT will be returned unless if the old one was logged out.

JWT's should be created with a login and reused by the caller until it expires. Tokens should not be created unnecessarily - e.g. do not create a new JWT for every API call.

Identity/Password + TOTP

Eclipse authentication allows admin users and customers to include multi factor authentication over and above identity/password This can be set on new or existing

identities and adds an additional RFC 6238 compliant one-time password requirement when authenticating. By simply creating or updating an identity and setting totpEnabled=true, Eclipse will respond with text which can be encoded as a QRCode and loaded into any RFC6238 compliant authentication app (E.g. Google Authenticator, Microsoft Authenticator etc). This then forces subsequent authentication requests to provide the 6 digit one-time password for the user. This password changes every minute and can only be accessed by someone who has the App and enrolled the QRCode in the App at the time of creating the identity. This is the identical principle to MFA authentication on platforms such as AWS.

Identity/Password + PKI (And potentially TOTP as well)

This authentication mechanism should be used for tenant system level access and for customers with large balances or wallets which will only be accessed with system API's.

The PKI authentication uses pairs of public/private keys held by the tenant and Eclipse. The tenant has a keypair for each identity they wish to secure with PKI, while Eclipse has a single keypair used to prove to clients that they are communicating with the true Eclipse API (and not an impersonator).

In order to authenticate with PKI, the flow is as follows:

Prerequisites

1. Client generates a 4096bit RSA public/private keypair as follows and stores the private key in a secure location and never exposes it to anyone (not even Eclipse):

   ```
   openssl req -nodes -x509 -sha256 -newkey rsa:4096 -keyout "PrivateKey.key" -out
   "PublicKey.crt" -days 99999
   ```

2. Client extracts the public key and sets it on the identity they wish to use PKI with (i.e. when calling POST on /eclipse-conductor/rest/v1/tenants/{tenantId}/admin-users or /eclipse-conductor/rest/v1/tenants/{tenantId}/customers/{customerId}/identities) The base64 encoded public key can be extracted from PublicKey.crt as follows:

   ```
   openssl x509  -in PublicKey.crt -pubkey -noout | grep -v "\-\-\-" | base64 -d| base64 -w0
   ```

3. Client has requested and received Eclipses's public key out of band (e.g. sent by email by Eclipse support)

The sandbox Eclipse public key in base64:


```
MIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEA2cJSqdgRyq2mvH0I0pg+pEBTKvuiQc/Bz2jIv8gvqXEx7l8AnBgaq
Fi53WnaCpumYF0+xjEqvfjlSWQGv6pkI/ZFCRJtZsehEvJ0DUo4qqByBgRCf28yGCSNxq3Nr/EFEqR+rmwSeegFo1x/QgtHzd
OT/32EI0i19vY1eVARM1f2YWfHxZR+zP6g+pm2RKI8cQ01udez4MMB/rhyy3id0/BAk/x/kx3+Y2qyKqqfhTSqjhs/gqnXg9V
TvizMzyJoxQZs+jgJJmbw+7WssstdvbqLe7+lJVgpDUdBs3768UPlaoYPDBkWtnS7U7W6zFHSsYu/wk3aYQIMKZqq/vZmBqzV
65oajK9Bt2viCRj5RvNClshQW+OLSeIkUrna5Xq1YkhYJTHDHd5qfaOdBDxFXFscfKkn9CM1LZT/siLeNkuBPTpaoRR+Wtp1S
heUIc+IC0f3CSYZrwRnuOn2tnGPbkLkEnDPxyx6sU97MEqjtQwgNPZTRI6YD4D4o/NRcmJ76OOtIJuoaMNoWv8p4FIFJ2AxVw
QVxRav9LEcByIEZukrMU1wTRnRvhfQgjG3idbg3XiUeTYdgGg1XjYB39JJ16w7cv4QW+J1kpl3BGL+ZgzlIiBZ8vd9Ou/4Ft1
2PT5TqVd0z4yciiZRL5eDHxn+KiM5GbRzapczI/jEA+vDutcCAwEAAQ==
```


The production Eclipse public key in base64:


```
MIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEAt20dRiY7sKgm/dLyjM6o1/Mm+vLlvx2yX1W5hRtQwIZ8bHZUrMita
M5tQJ9uuZ3wpqYSvz7CJ8FxAtevSSGYj345uk/rrHCT0xW64IzZbNeIcI19QowLKItx1PkmOb8MR/SfMoc/JUg8WuM/AaaBmw
OLGFk14+KI5v5+nCgLFoyZdUSuBVCvww1tJV694JblXbAhZLwJqeEXabnrIox/P956Au/rK3LqTpiwISg930lck8q6hjRIu76
QQjU2U0UYTxcm+eRDZ0eiqRLF7ZNrzIBfYCOf5X2okzW95Du0yqLK/WqVWOvj5kYcSdc0IXBg7Gvojz/wNZbM6GPD5x5RL86u
Oe5IqwnYogTc4UpoCnwtMF9MlpIPyu0ykD4kfspwWZQnSe5LrlkstlW22ZpoZFmqin4lNBwHfNQeGLb9HabiGCJ0XTTgswh0l
OXnu6sB9I1mH7h0RBVAqas84rRNVRZwQFYLqQFyRphPt9rGUT+RgG9NMnBVkEUAis+p2dM8q1MrQfuzmZnS1EhnSLevfJxDFQ
I+tK4d2Bo4P2wbWhPF7YfhOJq3MmuKyvsAS2hDtix0w6mUrSqHKsGA+1YAE4+NILpriMeAR8aICRabxcBl8bJywuNuHYcfZ4I
BZoUWG4ZUcLWudTEQOrKGSypoPcIU6WrFAswPpXHJJ1N/gPMCAwEAAQ==
```


Authentication Flow

1. Client calls GET /eclipse-conductor/rest/v1/authentication/login-challenges passing 2 query parameters: the identity they want to authenticate with and base64 encoded data called clientChallenge. clientChallenge is generated by the client by generating 64 bytes of random data and encrypting it with the Eclipse public key using RSA/ECB/OAEPWithSHA-1AndMGF1Padding and then base64 encoding the result. The client should store a SHA256 hash of the pre-encrypted 64 bytes so that it can verify the response sent by the server.
2. Eclipse receives the request and decodes and decrypts clientChallenge using Eclipses's private key and does a SHA256 of the result and base64 encodes that and puts it in response field base64EncodedClientChallengeResponse. Next Eclipse generates its own random challenge and encrypts it with the public key stored against the identity passed by the client and puts the base64 encoded result in field base64EncodedChallenge. Eclipse stores a SHA256 hash of its pre-encrypted challenge in order to verify the client's response to the challenge coming in the next step.
3. The client checks that the server's base64EncodedClientChallengeResponse is correct and if so knows the server must be in possession of the Eclipse private key. Assuming its correct (if not stop immediately), the client decrypts base64EncodedChallenge and gets a SHA256 hash of the decrypted data and puts this as base64 in a field called base64EncodedChallengeResponse in a body soon to be posted to the /eclipse-conductor/rest/v1/authentication/login endpoint. It also

does a SHA256 hash of the challenge it was asked to decrypt so that the server knows what challenge this is in response to. This is put as base64 in field base64EncodedChallengeHash. The identity and password fields are populated as per the simple identity/password authentication detailed earlier. This is posted to /eclipse-conductor/rest/v1/authentication/login (**This must be done within 600s of step 2 on sandbox and 60s on live**)

4. The server receives the login request and verifies that the client's response to the challenge is correct as well as the identity/password being correct. The server thus knows that the client must be in possession of the private key for the public key it has on record for the identity being authenticated. The server responds with a JWT as before.

Java Code Example

```
package com.mycompany.crypt;

import java.security.KeyFactory;
import java.security.MessageDigest;
import java.security.SecureRandom;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;
import java.util.Base64;
import javax.crypto.Cipher;

public class Main {

    public static void main(String[] args) {
        try {
            if ("generate-challenge".equals(args[0])) {
                generateChallengeAndExpectedResponse();
            } else if ("generate-response".equals(args[0])) {
                decryptServerChallenge(args[1]);
            } else {
                System.out.println("Mode can be generate-challenge or generate-response");
            }
        } catch (Exception e) {
            e.printStackTrace();
```

```
        }
    }


    public static void generateChallengeAndExpectedResponse() throws Exception {
        // Get from your own configuration somewhere. This will have been sent to you out of band
as per prerequisite 3
        // E.g. This is the Eclipse sandbox public key hardcoded as an example
        byte[] x509PublicKey =
Base64.getDecoder().decode("MIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEA2cJSqdgRyq2mvH0I0pg+pEBTK
vuiQc/Bz2jIv8gvqXEx7l8AnBgaqFi53WnaCpumYF0+xjEqvfjlSWQGv6pkI/ZFCRJtZsehEvJ0DUo4qqByBgRCf28yGCSNxq
3Nr/EFEqR+rmwSeegFo1x/QgtHzdOT/32EI0i19vY1eVARM1f2YWfHxZR+zP6g+pm2RKI8cQ01udez4MMB/rhyy3id0/BAk/x
/kx3+Y2qyKqqfhTSqjhs/gqnXg9VTvizMzyJoxQZs+jgJJmbw+7WssstdvbqLe7+lJVgpDUdBs3768UPlaoYPDBkWtnS7U7W6
zFHSsYu/wk3aYQIMKZqq/vZmBqzV65oajK9Bt2viCRj5RvNClshQW+OLSeIkUrna5Xq1YkhYJTHDHd5qfaOdBDxFXFscfKkn9
CM1LZT/siLeNkuBPTpaoRR+Wtp1SheUIc+IC0f3CSYZrwRnuOn2tnGPbkLkEnDPxyx6sU97MEqjtQwgNPZTRI6YD4D4o/NRcm
J76OOtIJuoaMNoWv8p4FIFJ2AxVwQVxRav9LEcByIEZukrMU1wTRnRvhfQgjG3idbg3XiUeTYdgGg1XjYB39JJ16w7cv4QW+J
1kpl3BGL+ZgzlIiBZ8vd9Ou/4Ft12PT5TqVd0z4yciiZRL5eDHxn+KiM5GbRzapczI/jEA+vDutcCAwEAAQ==");
        byte[] randomData = new byte[64];
        SecureRandom.getInstanceStrong().nextBytes(randomData);
        String expectedResponseFromEclipse =
Base64.getEncoder().encodeToString(MessageDigest.getInstance("SHA-256").digest(randomData));
        Cipher oaepFromAlgo = Cipher.getInstance("RSA/ECB/OAEPWithSHA-1AndMGF1Padding");
        X509EncodedKeySpec publicSpec = new X509EncodedKeySpec(x509PublicKey);
        KeyFactory keyFactory = KeyFactory.getInstance("RSA");
        oaepFromAlgo.init(Cipher.ENCRYPT_MODE, keyFactory.generatePublic(publicSpec));
        String challengeToEclipse =
Base64.getEncoder().encodeToString(oaepFromAlgo.doFinal(randomData));
        System.out.println("challengeToEclipse (What Eclipse should be able to decrypt): " +
challengeToEclipse);
        System.out.println("expectedResponseFromEclipse (What Eclipse should return to the
login-challenge call): " + expectedResponseFromEclipse);
    }


    public static void decryptServerChallenge(String serverChallengeBase64) throws Exception {
        // Your private key created with:
        // openssl req -nodes -x509 -sha256 -newkey rsa:4096 -keyout "PrivateKey.key" -out
"PublicKey.crt" -days 99999
        // Then for the public key to set on Eclipse identity: openssl x509 -in PublicKey.crt
-pubkey -noout |grep -v "\-\-\-\-\-"| base64 -d| base64 -w0
        // Then for the private key to encrypt with: cat PrivateKey.key |grep -v "\-\-\-\-\-"|
base64 -d| base64 -w0
        // This is just an example private key. DO NOT USE THIS IN PROD. The corresponding public
key is:
        //
MIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEAwTWV08TJLo/y+N67DjlFh0RFO4oyULlF5oWBain5xWzycFvbDTdiP
JvyfdACNsnEBcssjz5zD5ZOOQrJrYRgI2MzkhxmiXUWJAndZOixk6dhQObKru46xSRyOnI97UxFE4oBG+BLlC/jQucc7SF3yG
UCzdbugd4poLpn0CcVPGMfcPoVKNgK2XmyZAmZO13vgZctyS9VHHWwB1sOEtIUr4zkcon1lBZRqBv49y0d/QMkQ6Aeqi/C3T4
VPqhCDJNvG7h8RUTbroQav5YSnNwf4j+2rE4U1QO2Sc/noMWK1Xd8V792Nx61itfITNGhZEenPPyzVb9n8G6AlXsx+Kk8WpJB
9zbBvn8yqcXAZR5cHUz5fpmGnCIs4t1LclOxoqsXyrEtwT1dGGi22QD9zAEhw97NjXqhAH3rR4Z75qp51zFWnLgm4IrC+Blu6
```

mA7cG8pbER7CU6y7+tEVGtYgRiGT16Wmi4JKrjOlxcup+uQ+N+rBA8nfN3WP28Va6W0aXoWMnEao2PtHv3UYj7+TWZ/xDXb88
D6fh+DngBHeI5axuh++Z74akXYzD9frsDKk9y/wWQv2a20HftQIlVMLYu84ZBUnyQj7VxqYQ2yuNu9jJKY38SlAlgWKgAQF6o
wtG8YUFdGgBGig/Odd9bU94SGwwi24LnT2mSsN6O+OyUo+lcCAwEAAQ==

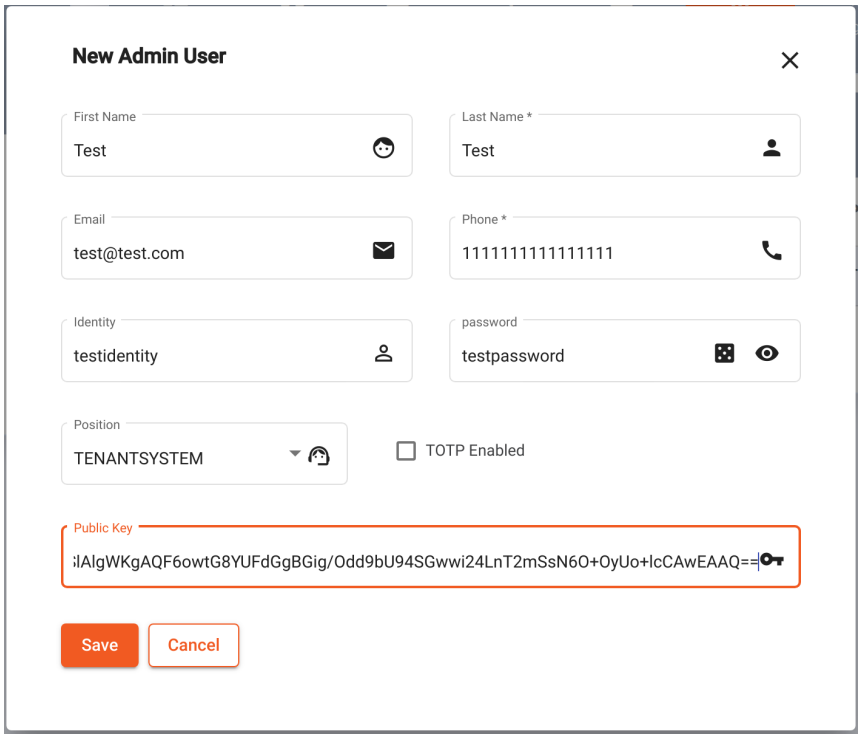
```
        byte[] pkcs8EncodedPrivateKey =
Base64.getDecoder().decode("MIIJQgIBADANBgkqhkiG9w0BAQEFAASCCSwwggkoAgEAAoICAQDBNZXTxMkuj/L43rsOO
UWHREU7ijJQuUXmhYFqKfnFbPJwW9sNN2I8m/J90AI2ycQFyyyPPnMPlk45CsmthGAjYzOSHGaJdRYkCd1k6LGTp2FA5squ7j
rFJHI6cj3tTEUTigEb4EuUL+NC5xztIXfIZQLN1u6B3imgumfQJxU8Yx9w+hUo2ArZebJkCZk7Xe+Bly3JL1UcdbAHWw4S0hS
vjORyifWUFlGoG/j3LR39AyRDoB6qL8LdPhU+qEIMk28buHxFRNuuhBq/lhKc3B/iP7asThTVA7ZJz+egxYrVd3xXv3Y3HrWK
18hM0aFkR6c8/LNVv2fwboCVezH4qTxakkH3NsG+fzKpxcBlHlwdTPl+mYacIizi3UtyU7GiqxfKsS3BPV0YaLbZAP3MASHD3
s2NeqEAfetHhnvmqnnXMVacuCbgisL4GW7qYDtwbylsRHsJTrLv60RUa1iBGIZPXpaaLgkquM6XFy6n65D436sEDyd83dY/bx
VrpbRpehYycRqjY+0e/dRiPv5NZn/ENdvzwPp+H4OeAEd4jlrG6H75nvhqRdjMP1+uwMqT3L/BZC/ZrbQd+1AiVUwti7zhkFS
fJCPtXGphDbK4272MkpjfxKUCWBYqABAXqjC0bxhQV0aAEaKD85131tT3hIbDCLbgudPaZKw3o747JSj6VwIDAQABAoICAALT
A7Oqxr4VgJm+t8FHeQq3JsPuCzo+2xS5kuC4aFxIcWrPDO6uhpMYPH02UzH11RW4mJgnc4sgyVfwHm6Y5sbdpl1WaPFNcttoA
YJ+HEpqQfOu6/uOo/FJzRmpndogKUpV1BpSNwidgpbxxex5dsTJJTdZEvd3BejBkM7sI1VUpZsDLNgCAHaZXtAzi7fgcaXVTe
gI9xeJaN9Sp2wDAiZmytgZyCs4AE2Syxrh3/RY3WjXqFsYg6K0S5802nJQnWksUzS4AvYUHEf8q5z7YsPt5n+yTH+opDuD/ID
PIm5oZrrpo5K+siyrZtQopFFQ8tLjZsO0HpLXKq4EtQBLiQRLWzKQu7Y0z1KTx4jss8XCay18QSryBZOsAgiC1Vd0tG5BdG/Q
xNw0wA59N4TtaScvSGklh99bwHwYkoX2EFrpqQeXa4G4BT1b9grVMddAKCdY76biOSqLarpSX1363vEfAj6FqkaKUPIfYU7Hc
WrpicdSNlmD5pP+cAeKCshyBsu7V83AqJIMixVRVw+BE/TEXMSdaRsf5FWy6sil85dw/+8ztbKJ1AfC/gcpVYia/OVkKnNvKP
5L7YHJM4YH1TZ0Z6WmVsX+oKoJv06ysmVH24zlItVf4ZiIj+x/4bbgaGztclpT3QYHfAnALU1XJajU7gANxGX9puOHGe0bJ/t
hAoIBAQD+ralLGA24igFNPeoTDekeisZWoyxFWi+9/LutZGgkA56n+lD7xl5+EC7AXAtnBm6fUDsw+nBPxhLDq/C40Et2m859
6U1j5bhDIuXNhK42QUYeE0vsjjoGrPjaEBIhlClLbK3BJNBSV8u8zOCfyisZdY4AdDPG1gFvtkQSEBqqMJATAKuPuEZHdLbn1
16xANEPzLqBAOZ1qPBzEdysN+3rivdeGdEtp7MdxKa7xJWdCRIlYopZdMLDqU3n35kc3D6NnNHjSqulntICcZ3J8rXRSzUlnT
LRNiLE90b56VlI6esN/B2ZeIzqgwkF7PTnMRIzTmIWWUi3lOvy3UpFzh8ZAoIBAQDCNkNACAla05PeBk0EtLG5rDTwaLpeBQU
A7DyQbcCpUE7OOgv+EWNwuhu/0/OpDL5q40k3vddcryJOrAKdIpqwlzKVVmO2X9H0Ysl0EgezdE56C9zy9xPMvE3pjL+aPYID
TMm15ebNNxPzYs4LDlMhaiRKHKTyojUsyx541VjAEP0vuos2dVNWZ7kfd/FYHKcsEAgId5QwreTMrfm/3hDyblfcJzdq4Qptj
aEKt3IVoCLvvvyYAgpFELoM37DuRyx3Mz6PTJupQgcvPLYO0+4z/MOOiQKki3mPatYVxDbGKtUFEtzMx6fNOitrFH+OGl9wtb
oGOwjYJWkWkcmRkcLvAoIBAQDTO4yf2TqAJg7AVysPVMlW6Llq4Hn2p7/Tb4zmWtI5lLsCDg8fVL/JBXkRWQaoE+CqyKvfisi
ekeW8X58QWstMfBw/uMLUqZM0M/ywgmzQWUFj7pNZh/m+0baxfwTazv2VLnXLLdKPiXxVgwSH4/HtcuAtpjPPpm6Yjm9VIABN
ymulJzgkwy2fVMHj93D1lHzSAvdpyipbypxshJMoUFdItwFHDli96ksY1KOKN5n0HrLrSRkA62QjTwhr2rwvAMntQPWg9Kz7e
IgtOLkr5+A/0IYrRYABoEokXgiDfbTtUf8AoWyXfzI5zCcuwAO8wamb3lZivPkgMyekT0CsNzL5AoIBAHaTxu+7eGq6L9gppL
Cw2qpw2EnIx7FMA0gS9M+CE1XNMLB9hID1gO980iCHeofGviRo7XVZNbrjO2EpaNckH6qeRkYj3vGlA33hKvrxXz5UTu35fEf
kT3R7rF5ij0HPOChno+ipnAfEDyxVMVRSxLLsKYgOgS+bljfpNIu2R1/Nf6i6jvELJXzmC4OJKhQQOomOqDjzOcmXAtFtj8V0
AojikkZas11QVZuaE6+FXeSSR4J166er5zVQByiBE3gIgL+uzA2WSpxHMjfHK6lWAGRBeez2/mQUvH2yT1Au4vqPgEx+vc1SQ
cPuNkBE3bSXqZoZeqD/PmEqRkV/ccCtbIUCggEAcrAQWZNzlGrpDHYUYZilHpUtGzU7/utQomhoBw29Di4FbJHD2gHEGd2flC
LdXwe9HRj/cVCOagNcH7jjhfRt3lup8ogZaG6n+8NvQUJxtbP7JGfdmMNT06Y0mQk6kIp1+gMeMJ/ECCHgxdlgYwcJbDxQ/eC
FuQUpM0vElkiqBr4ioBZgPEuIWjLNuKJ314W8BiGXQTH5vDnhPcGhOWSPaNHVGHKbZC2EpBRq5tXK2BylkFJxT6AUfD51N8Mb
BsS1fl95AUijKtaPjEJ7WFzvnXmncrKLC+HY7wb9qAc3X3BlViVMffmn6nDK6AtEc8PnUtLqqjsdj/OCy4jiCezL2A==");
```

```
        PKCS8EncodedKeySpec privateSpec = new PKCS8EncodedKeySpec(pkcs8EncodedPrivateKey);

        KeyFactory keyFactory = KeyFactory.getInstance("RSA");

        byte[] serverChallenge = Base64.getDecoder().decode(serverChallengeBase64);

        Cipher oaepFromAlgo = Cipher.getInstance("RSA/ECB/OAEPWithSHA-1AndMGF1Padding");

        oaepFromAlgo.init(Cipher.DECRYPT_MODE, keyFactory.generatePrivate(privateSpec));

        byte[] decryptedChallenge = oaepFromAlgo.doFinal(serverChallenge);

        String base64EncodedChallengeResponse =
Base64.getEncoder().encodeToString(MessageDigest.getInstance("SHA-256").digest(decryptedChallenge
));

        String base64EncodedChallengeHash =
Base64.getEncoder().encodeToString(MessageDigest.getInstance("SHA-256").digest(serverChallenge));

        System.out.println("base64EncodedChallengeResponse: " + base64EncodedChallengeResponse);

        System.out.println("base64EncodedChallengeHash: " + base64EncodedChallengeHash);


    }
```

```
}
```

Here is an end to end example done with the above code on sandbox:

1) Create admin user. Here we use the same public/private key as per the previous example:



2) Run java program to generate login challenge request:

```
challengeToEclipse (What Eclipse should be able to decrypt):
1TGcy5l3ZnmndwWPGk1s++IqVMOpOx46AH+8MgTIX07byn485agyYIdLsNaHyph4K/gDfjeXdbc+fyOCkqAaQYsMhhE5ttcAm
BEuX0NueEXDccCoJ9A3IwGEzOzqHZjp3egiYrsU5Jn23nDT1XHMSrRY21p1DdilwnoYlnOXO+uRf8xRSrZLKUekPx43O6Rpy/
AShRl2gwQug9zgAap40blT6aSEIL27bq0zzk9VJoeiQfncL7P3OM/kI3eP146enUuML8pmyqnUf0NX9lkkXLO4rz9ZCdw/UZH
Micuvc2Jcwt4AEDnxZlbQLT6eYptMqzg6XcbgOfZvC2fKOGhN0rfgZVw81/e6UjC6ZipBkco2jqpwSgh6qeqy2MnGvg/VTtD7
RNUT9VigqLiIDTDraD2y5sdXtP8SewcfDFX4eUUAzq2dG9yb6//5w9TmkdefSwFUKqJsqQldpOF0iHI9cvzaTgidsJwrZzgGt
XYewm18cRRGvtzdYkl7+YQje9AtnwRmSKse6Z068YALqxDvxBbK0TLa8pS+O0j0GfJI3hLS9DAGA5NS/8UIp9/Yq3S8HxJUnP
0k5SZd8K4pY9xb9TzQOoyGNy/g1qjzC2tAe0+aw0eMxFAlwtyzIw5A2PYkfW5NdSXTaz5cEDpelBrRfGNdJ+KGwl0fLM5CIbY
+MbQ=
```

```
expectedResponseFromEclipse (What Eclipse should return to the login-challenge call):
DeAObN6HXiBfXPNEczTpEmAJxTwakG/RSGOe0w8+e8U=
```

3) Send request to get login challenge:

```
curl -X GET
"https://eclipse-java-sandbox.ukheshe.rocks/eclipse-conductor/rest/v1/authentication/login-challe
nges?clientChallenge=1TGcy5l3ZnmndwWPGk1s%2B%2BIqVMOpOx46AH%2B8MgTIX07byn485agyYIdLsNaHyph4K%2FgD
fjeXdbc%2BfyOCkqAaQYsMhhE5ttcAmBEuX0NueEXDccCoJ9A3IwGEzOzqHZjp3egiYrsU5Jn23nDT1XHMSrRY21p1Ddilwno
YlnOXO%2BuRf8xRSrZLKUekPx43O6Rpy%2FAShRl2gwQug9zgAap40blT6aSEIL27bq0zzk9VJoeiQfncL7P3OM%2FkI3eP14
6enUuML8pmyqnUf0NX9lkkXLO4rz9ZCdw%2FUZHMicuvc2Jcwt4AEDnxZlbQLT6eYptMqzg6XcbgOfZvC2fKOGhN0rfgZVw81
%2Fe6UjC6ZipBkco2jqpwSgh6qeqy2MnGvg%2FVTtD7RNUT9VigqLiIDTDraD2y5sdXtP8SewcfDFX4eUUAzq2dG9yb6%2F%2
F5w9TmkdefSwFUKqJsqQldpOF0iHI9cvzaTgidsJwrZzgGtXYewm18cRRGvtzdYkl7%2BYQje9AtnwRmSKse6Z068YALqxDvx
BbK0TLa8pS%2BO0j0GfJI3hLS9DAGA5NS%2F8UIp9%2FYq3S8HxJUnP0k5SZd8K4pY9xb9TzQOoyGNy%2Fg1qjzC2tAe0%2Ba
w0eMxFAlwtyzIw5A2PYkfW5NdSXTaz5cEDpelBrRfGNdJ%2BKGwl0fLM5CIbY%2BMbQ%3D&identity=testidentity" -H
"accept: application/json"
```

{"expires":"2021-01-28T16:07:13.020278683Z","base64EncodedChallenge":"dL2TobNg1K6ISkw2L80zH8c6E7K
7yQNndHDA/ntKuw/K6k0IsIbEUfkwQf5vUwNQSjApSne5QORXx3ZCFe4BroP3HRLxR7v3dPhYiSBbZGPP3XMdfq3Zm4eztSVc
Ntv+JZOiAJvghbO4hvqHy8rjvEFBTAhqlcMB6wejpvZBxdd3cqQCwrCACrgQ2GzRvo1nA505qVyB/2v7IpX/AXyjHxiqX9PZx
Jz9gLdRSHNllL6mrjWL9B8ldvCjBwdk0nYc1zfX46Lw1/2kdsdZYMkOsiFwy5E9lYSBssdYAXzRmuwkko16vSiw4BPEb3VZl4
WpGkjtvgnEpnqAGlTq+fuVXwl4fDEG53OxkNOn0uLGV3s6hlgtkYdpqOzmy10g1wIHuzEZo3q7SXpBL+mQC+NWS2uFOE9RFDo
lI+JGwk6ZVknNGeCalv7wwF+d6eQD5Ip8VRKD72d1Ce8g6JJYp3viUWDMEPbz0HNkos4NQlLG3uxLovXn/Jq/b+9Aygw3jIMH
XDin6ETkkymqfNdmqjoC0jpl07kxVK+Ny3NLE7jYoKFmAT6UNodGOwC4TSdWu6ebL/4SNAVThC4ZzaLtnTypSRSRhDWEbUj63
478RgSE+z3Iy+paChwhSr8q3VnyPMrFzsrsvjxUL4uXVqWAF44oP7PO+bW0AkwUcsGgZVS+LeE=","base64EncodedClient
ChallengeResponse":"**DeAObN6HXiBfXPNEczTpEmAJxTwakG/RSGOe0w8+e8U=**"}

4) Verify DeAObN6HXiBfXPNEczTpEmAJxTwakG/RSGOe0w8+e8U= is correct

5) Pass base64EncodedChallenge to the java program:

```
base64EncodedChallengeResponse: wadU7UwGsjY2Zs10mbqZxLNvy/EjPSLdb7WmwdpAnwc=

base64EncodedChallengeHash: jNyQxcZIXgvuzDOfEOQQ1ACisGQ2uKa8RqG4/JWPUCM=
```

6) Login using the challenge response and hash and identity and password:

```
curl -X POST
"https://eclipse-java-sandbox.ukheshe.rocks/eclipse-conductor/rest/v1/authentication/login" -H
"accept: application/json" -H  "Content-Type: application/json" -d
"{\"base64EncodedChallengeHash\":\"jNyQxcZIXgvuzDOfEOQQ1ACisGQ2uKa8RqG4/JWPUCM=\",\"base64Encoded
ChallengeResponse\":\"wadU7UwGsjY2Zs10mbqZxLNvy/EjPSLdb7WmwdpAnwc=\",\"identity\":\"testidentity\
",\"password\":\"testpassword\"}"
```

{

  "headerName": "Authorization",

  "headerValue": "Bearer
eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJ0ZXN0aWRlbnRpdHkiLCJ1aWQiOjMxLCJwb3MiOlt7Im8iOjE1LCJkIjowLCJwIjoi
VEVOQU5UX1NZU1RFTSJ9XSwiY2giOiJHUzB5Q2RCd25NIiwicm9sZXMiOiIiLCJzZXNzIjoiYzA1NmIyZDgtZDZkMS00ZWE2L
Tg3NTItZTg3ZjY4NGIyOTAzIiwiaXNzIjoiaHR0cDpcL1wvamluaS5ndXJ1IiwiZXhwIjoxNjExODUxNDEzLCJ0ZW5hbnQiOj
E1fQ.j1QDr6G2TdraAAJFum6IVenFzeUfJteoaiD-bq134mY",

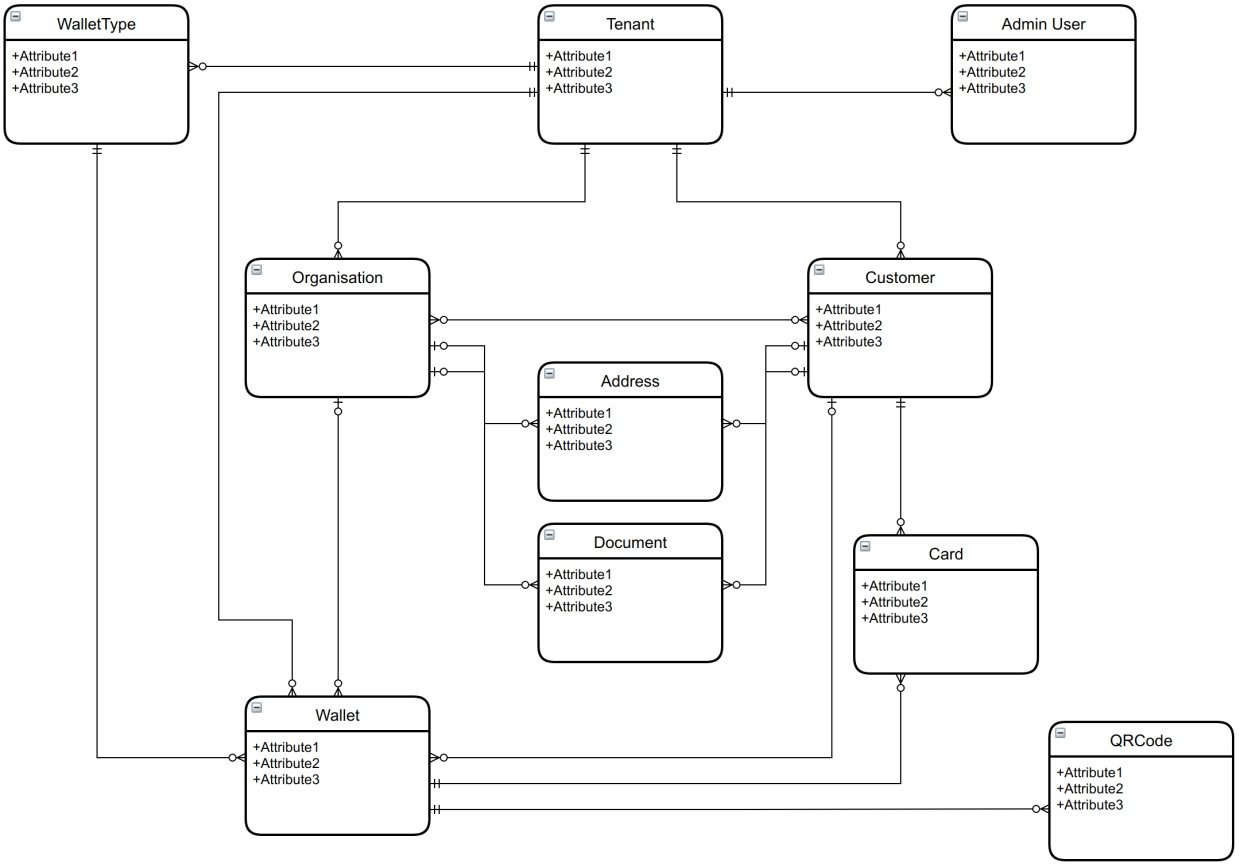
  "sessionId": "c056b2d8-d6d1-4ea6-8752-e87f684b2903",

  "expires": "2021-01-28T16:30:13.341Z",

  "expiresEpochSecs": 1611851413

}

High Level Entities in Eclipse



High level Entity description

A Tenant is the contracting partner with Ukheshe and responsible for the management of their wallet programs and taking a fintech proposition to market. Depending on the scenario, a company could set up multiple tenants on Eclipse if they have different propositions with different goals and target markets.

A Tenant has to be created before any other entity can be created or edited. It is the cornerstone of data segregation. A tenant can only access things within the tenant. This is enforced at multiple layers by Eclipse.

A Tenant can have multiple customers and organisations under itself.

A tenant can have admin users who can manage/administer Eclipse for the tenant.

Customers and Organisations have addresses and documentation attributes to be used for KYC and reporting purposes.

Tenants, Customers and Organisations can have wallets. Depending on the Wallet Type, a wallet can have an associated card (Physical and/or Virtual) and/or QR Code.

Data Segregation

The data in Eclipse is stored in a geographically redundant distributed database where granular authorisation rules ensure that the API caller can only CREATE/READ/UPDATE/DELETE data accessible to the caller. A fundamental rule that is checked at multiple layers ensures that a tenant's data is only visible to that tenant and/or global users such as Ukheshe administrators and support staff. This is all controlled via the granular access permissions configuration set up per tenant.

Here is an example of a part of the permissions configuration:

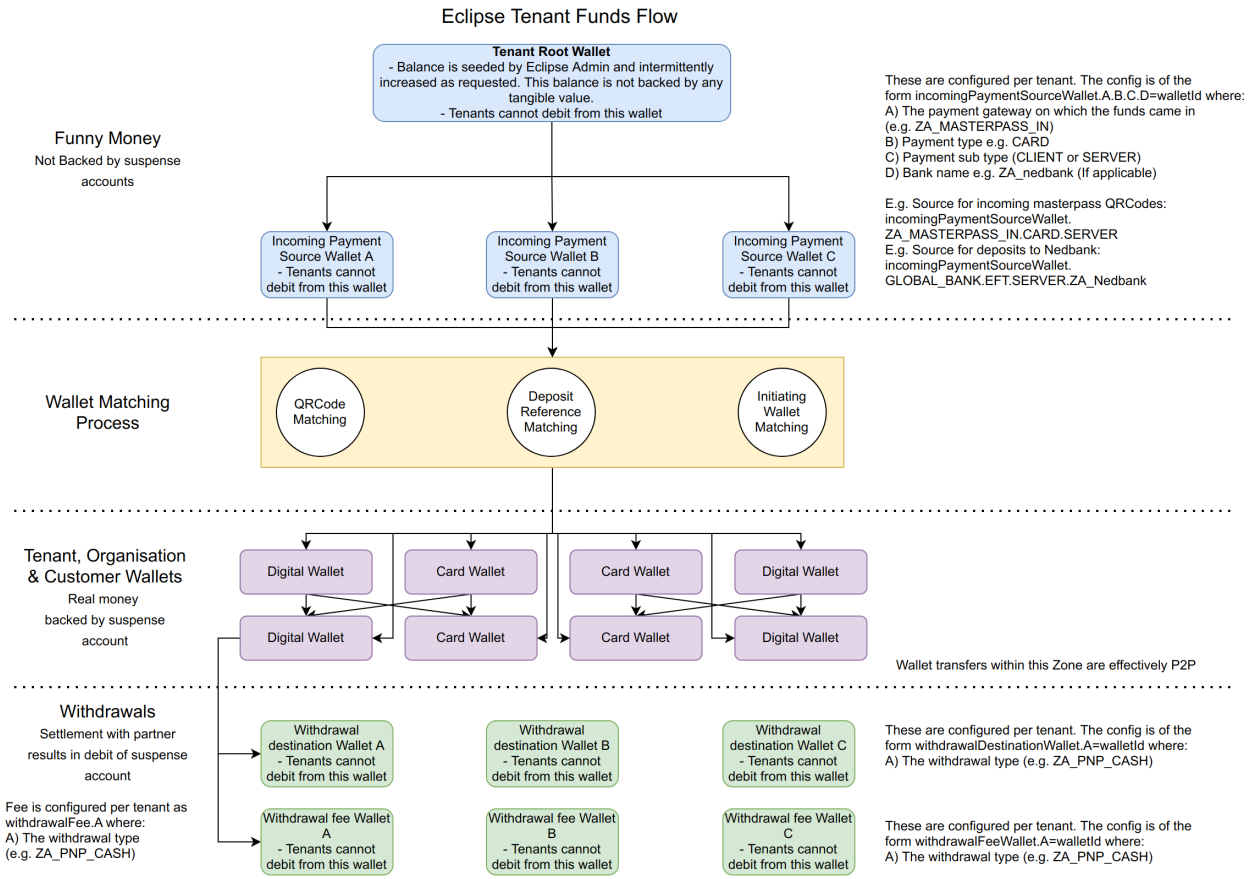
Copy the block to the right to use as the tenants permissions configuration	Transfer.WalletType27.CREATE.Allow TransactionDescription.WalletType27. Wallet.WalletType27.CREATE.Allowed Wallet.WalletType27.UPDATE.Allowed	Global Roles (NA Means its there by default)								Tenant Admins											Position in Org Owning the Entity				Associated Custo	Anon
Permission	Description	GLOBAL_ADMIN	GLOBAL_FINANCE_ADMIN	GLOBAL_FINANCE_CLERK	GLOBAL_TECH_SUPPO	GLOBAL_SUPPORT_L1	GLOBAL_SUPPORT_L2	GLOBAL_SUPPORT_L3	GLOBAL_INTEGRATOR	TENANT_SYSTEM	LEVEL_01	LEVEL_02	LEVEL_03	LEVEL_04	LEVEL_05	LEVEL_06	LEVEL_07	LEVEL_08	LEVEL_09	LEVEL_10	HEAD	MANAGER	EMPLOYEE	MEMBER	CUSTOMER	ANONYMOUS
Payment.UPDATE.Allowed		Y	Y	Y																					Y	
Payment.READ.Allowed		NA	Y	Y	Y	Y	Y	Y		Y	Y														Y	
CardOnFile.CREATE.Allowed		Y								Y															Y	
CardOnFile.UPDATE.Allowed		Y																							Y	
CardOnFile.DECRYPT.Allowed																									Y	
CardOnFile.READ.Allowed		NA	Y	Y	Y	Y	Y	Y			Y														Y	
CardOnFile.DELETE.Allowed		Y																							Y	
Customer Management																										
Ratify.CREATE.Allowed		Y				Y	Y	Y		Y	Y														Y	
Ratify.READ.Allowed		NA	Y	Y	Y	Y	Y	Y		Y	Y														Y	
ManualRatify.CREATE.Allowed		Y				Y	Y	Y		Y	Y															
UserIdentity.CREATE.Allowed		Y								Y																
UserIdentity.UPDATE.Allowed		Y								Y																
UserIdentity.READ.Allowed		NA	Y	Y	Y	Y	Y	Y		Y	Y														Y	
UserIdentity.DELETE.Allowed		Y								Y																
UserPosition.CREATE.Allowed		Y								Y																
UserPosition.READ.Allowed		NA	Y	Y	Y	Y	Y	Y			Y										Y				Y	
UserPosition.DELETE.Allowed		Y																								
UserPositionInEmptyOrganisation.CREATE.Allowed		Y																							Y	
Document.CREATE.Allowed		Y				Y	Y	Y		Y	Y										Y				Y	
Document.UPDATE.Allowed		Y				Y	Y	Y		Y	Y															
Document.READ.Allowed		NA	Y	Y	Y	Y	Y	Y		Y	Y										Y				Y	
Document.DELETE.Allowed		Y								Y															Y	
Address.CREATE.Allowed		Y								Y											Y				Y	

This data segregation allows not only for a tenant to ensure that their data is theirs, but also that within the tenant, they can control who can see/do what. This helps ensure that for example customer A on tenant 10 cannot see customer B's data even if they are also on tenant 10.

If a tenant wishes, the permissions can be set so that not even Ukheshe staff can see their data (other than GLOBAL_ADMIN which is required for some severe operations tasks). This would however mean that Ukheshe cannot support the tenant with queries via the admin portal as they would not have access.

Ukheshe considers a tenant's data to belong to the tenant and does not distribute, share, extract or otherwise utilise the data for any other purpose other than to support the tenant in their business operations and billing the tenant for Eclipse usage.

Wallet Hierarchy & Fund Flows



In essence Eclipse has a concept of wallet types which can be set up under a tenant. One specific wallet type is a SYSTEM wallet which has rules that it can only be debited by the system itself. These wallets are used as the source for incoming funds and the destination for outgoing funds.

Non-system wallets such as digital wallets and card wallets hold funds which are effectively backed by real money in a suspense account at a bank. The sum total of money in these wallets (the purple area above) would equal that in the tenant's suspense account (other than slight timing differences). Funds flowing into the suspense account via topup mechanisms result in a debit of an incoming payment wallet (blue wallets) and a credit of a purple wallet.

Withdrawals from purple wallets result in a debit of the wallet and a credit of a withdrawal destination (green wallet) and possibly of a fees wallet.

Card spend in purple wallets results in debits off a card settlement suspense account which is in turn kept at a float by debiting the suspense account.

The source and withdrawal SYSTEM wallets are configurable and can be the same or different for each topup/withdrawal type. The ledger of these wallets is available to the tenant for reconciliation purposes but tenants cannot debit from them. The balance in these wallets does not represent real money but the balance in source wallets must be sufficient to cover incoming funds. This acts as a safety net to ensure money cannot enter a tenants environment without having followed a 2-step process - first from the root wallet to a source wallet (done by Ukheshe finance administrators) followed by a transfer initiated by the system when a topup occurs.

For an overview on the flow of funds (real money) refer to the Finance Flow of Funds document here FINANCE FLOW OF FUNDS

Use Cases

The following sections describe the usage of the most important ReST API's in terms of use cases from the perspective of a tenant. If a use case is not covered in enough detail then please contact your Ukheshe account manager for more detail.

Initial Prerequisite - Tenant onboarding

Create a Tenant

All API's on Eclipse are called on behalf of a tenant. Either by the tenant itself (tenant admin users) or customers belonging to the tenant. The first step in using Eclipse is to onboard as a tenant. This is done by Ukheshe staff. If you would like to have a tenant created on Sandbox then please ask your account manager to do so. This is a quick process and you will be provided with an administration username and password and your tenant id.

Depending on requirements, the sandbox tenant will be created with the necessary system wallets and configuration to support the scenarios being implemented.

Set up Admin Users within the Tenant

It may be necessary to set up additional users (or API clients) within your tenant so that your API's and employees can access the Eclipse API or Admin portal. To set up an admin user within the Tenant, create an admin user with a POST:

Prerequisites

A valid bearer token for the system identity

Step 1 - Create profile for tenant admin user

POST /eclipse-conductor/rest/v1/tenants/{tenantId}/admin-users

Pass the relevant details of the employee

This adds the user which can then itself authenticate against the Eclipse API.

Create an Initial Wallet Type

The behaviour and functionality of wallets are determined by the wallet type. This is like a template for creating wallets. In order for tenants to create wallets, there must be at least one wallet type to specify how the wallets will behave. For sandbox it's normally adequate to create a closed loop digital wallet type with a single attribute "Currency" and value "ZAR". These wallets allow a minimum balance of 0 with no credit limit.

Additional wallet type configuration parameters that determine a wallets behaviour are as follows:

Parameter	Possible Values	Description
Currency	ISO 4217 Currency code	The underlying currency of the balance of the wallet. This is mandatory for digital wallets
ProgramCode	Pre configured program code in MPTS	Only applicable to wallets that back cards. Defines the configuration in MPTS for the cards linked to the wallet. This is mandatory for card wallets.
Kyc	Pre defined rule names set up by Eclipse support	If present then any wallet of this type created for a customer will check if the customer satisfies the rules for owning the wallet. E.g. what KYC documents need to have been submitted and whether their checks pass.
Kyb	Pre defined rule names set up by Eclipse support	If present then any wallet of this type created for an organisation will check if the organisation satisfies the rules for owning the wallet. E.g. what KYB documents need to have been submitted and whether their checks pass.
sendTransferLogicSetId	Numeric	Pre defined Id (numeric) defining a set of logic to run whenever a debit or credit happens on a wallet. This logic can include things such

		as additional transfers to do, checks or whatever custom logic is needed by the tenant for the wallet type. The logic can be updated at runtime and is extremely flexible as it is effectively code injected at runtime to run as part of the wallets transaction. Specific requirements for a tenant can be agreed with Eclipse Support and implemented for a wallet type.
monthlyFeeLogicSetId	Numeric	Pre defined Id (numeric) defining a set of logic to run once per wallet on the first of each month. This logic can include things such as monthly fees. The logic can be updated at runtime and is extremely flexible as it is effectively code injected at runtime to run as part of the monthly fee batch. Specific requirements for a tenant can be agreed with Eclipse Support and implemented for a wallet type.
minimumBalance	Numeric	The balance below which a wallet cannot be debited for normal payments and transfers. Can be used for fees. Only applicable to digital wallets.
walletMovementWebhookUrl	String	URL that Eclipse will do a POST to with the transaction data for every

		transaction that impacts the ledger for this wallet type. Note that this is only supported on wallet types where Eclipse is the store of value. E.g. does not do anything for prepaid cards as Eclipse is not the ledger.

Wallets in Eclipse are very powerful and customisable through plugins and extensions - if any specific use case is needed by a tenant then please discuss this with the Eclipse team.

How To Define Kyc/Kyb parameter on wallet type:

1. Use Kyc/Kyb as a config attribute key and any string as value in wallet type configuration.
2. Define the value of the Kyc/Kyb parameter in the property table.

For example if we have define “Kyc” as key and “SpecialRulesForWallet” as value in wallet type configuration then we must need to define rule set of “SpecialRulesForWallet” in to property table of eclipse like:

kyb.ruleset.SpecialRulesForWallet = public static com.ukheshe.eclipse.conductor.model.EclipseWalletAllowedResult kycRuleSet(com.ukheshe.eclipse.conductor.model.EclipseKycResult kycResult) {... implementation..... }

These rules will be invoked at runtime when a wallet is created using the specified wallet type.

Prerequisites

A valid bearer token for the system identity

Step 1 – Create wallet type

POST /eclipse-conductor/rest/v1/tenants/{tenantId}/wallet-types

Change Tenant System User Username and Password

Prerequisites

A valid bearer token for the system identity

Step 1 – Request a new tenant username and password

POST /eclipse-conductor/rest/v1/tenants/{tenantId}/system-identities

If the username is the same as before then this will result in the password being updated.

The above steps would result in the tenant being onboarded, having appropriate staff linked to the tenant and the tenant having a system identity for all core API calls and an initial wallet type for creating wallets.

Standalone Customer Use Cases

The following use cases are ones where there is no grouping of customers into organisations and no use of company cards and other functionality for corporate use cases. The simpler cases focus on a tenant selling to customers who have wallets/cards

Register a new customer with a digital wallet

In order to sign up a new customer and issue a digital wallet, first create the customer along with the required customer information and then create the wallet with a walletTypeId for a digital wallet mode e.g. CLOSED_LOOP_DIGITAL

Prerequisites

A valid JWT for API calls.

The customer should have KYC documentation available.

It is recommended to search for a customer to view any potential duplicates of the cardholder within the Tenant.
GET/eclipse-conductor/rest/v1/tenants/{tenantId}/customers

You will require the customers personal information (Name, ID, Address etc)

Step 1 – Create Customer

First create a customer providing all required fields

POST /eclipse-conductor/rest/v1/tenants/{tenantId}/customers

The system will reply with the customer along with its unique customerId

Step 2 – Add Address Details

Add address details for the customer (if required for the level of KYC required for the wallet type)

POST /eclipse-conductor/rest/v1/tenants/{tenantId}/customers/{customerId}/addresses

Step 3 – Add KYC Doc info

Add customer supporting KYC documentation (if required for the level of KYC required for the wallet type) There are several type of document that customer can upload for KYC e.g

FACIAL_PHOTO, NATIONAL_IDENTITY, PASSPORT, PROOF_OF_ADDRESS. We can also use document type AUTO if we want to let eclipse detect the type of uploaded document whether it is NATIONAL_IDENTITY or PASSPORT.

Note: Customer must not use COMPARISON_NATIONAL_IDENTITY, COMPARISON_PASSPORT, COMPARISON_FACIAL_PHOTO for KYC purpose as these document types are used for checking if the document matches what is already on file (The Typical use case for these document type are password resets etc)

POST /eclipse-conductor/rest/v1/tenants/{tenantId}/customers/{customerId}/documents

Step 4 – Kick off Ratify

POST /eclipse-conductor/rest/v1/tenants/{tenantId}/customers/{customerId}/ratify

This will kick off the various checks and algorithms in Eclipse to analyse the customer profile, address/s and attached documents to update the customers KYC status. This call will return detailed results of the KYC process such as what passed and what failed in the various tests. The customer profile can be updated along with addresses and documents and ratified again until the required KYC status for the wallet type is obtained. The wallet type determines what level of KYC is required for a customer/organisation to own that type of wallet.

Step 5 – Create Wallet

Create a digital wallet for the customer. This refers to the following API in swagger:

POST /eclipse-conductor/rest/v1/tenants/{tenantId}/customers/{customerId}/wallets

WalletTypeId must be of a type for a digital wallet mode.

Register a new customer and issue a virtual card

In order to sign up a new customer and issue a virtual card, first create the customer along with the required customer information and then create the wallet with a walletTypeId for a PREPAID_CARD wallet mode.

Note that as prepaid card wallets always have an initial card it will be created when the Wallet is created.

Prerequisites

A valid JWT for API calls.

The customer should have KYC documentation available.

It is recommended to search for a customer to view any potential duplicates of the cardholder within the Tenant.

GET/eclipse-conductor/rest/v1/tenants/{tenantId}/customers

You will require the customers personal information (Name, ID, Address etc)

Step 1 – Create Customer

First create a customer providing all required fields

POST /eclipse-conductor/rest/v1/tenants/{tenantId}/customers

The system will reply with the customer along with its unique customerId

Step 2 – Add Address Details

Add address details for the customer

POST /eclipse-conductor/rest/v1/tenants/{tenantId}/customers/{customerId}/addresses

Step 3 – Add KYC Doc info

Add customer supporting KYC documentation.

POST /eclipse-conductor/rest/v1/tenants/{tenantId}/customers/{customerId}/documents

Step 4 – Kick off Ratify

POST /eclipse-conductor/rest/v1/tenants/{tenantId}/customers/{customerId}/ratify

This will kick off the various checks and algorithms in Eclipse to analyse the customer profile, address/s and attached documents to update the customers KYC status. This call will return detailed results of the KYC process such as what passed and what failed in the various tests. The customer profile can be updated along with addresses and documents and ratified again until the required KYC status for the wallet type is obtained. The wallet type determines what level of KYC is required for a customer/organisation to own that type of wallet.

Step 5 – Create Wallet

Create a wallet for the customer which will automatically create and link a virtual card.

This refers to the following API in swagger:

POST/eclipse-conductor/rest/v1/tenants/{tenantId}/customers/{customerId}/wallets

WalletTypeId must be of a type for mode PREPAID_CARD

Do **not** populate the primaryPhysicalCardIndetifier field as this is for issuing physical cards not virtual cards.

You can now optionally call:

GET /eclipse-conductor/rest/v1/tenants/{tenantId}/customers/{customerId}/cards

This will return the card created for the customer.

GET /eclipse-conductor/rest/v1/tenants/{tenantId}/customers/{customerId}/wallets

This will return the wallets belonging to the customer.

Register a new customer and issue a physical card

In order to sign up a new customer and issue a physical card, first create the customer along with the required customer information and then the card handed over to the customer needs to be registered and linked to the customer.

Depending on the card packaging, either the PAN, Pack ID or Card QR code will be used to register the physical card.

You will need to create a Wallet with a card mode.

Prerequisites and Step 1 to 3 are identical to issuing virtual cards.

Step 4 – Create Wallet

Create a wallet which will automatically be created and link the physical card.

This refers to the following API in swagger:

POST/eclipse-conductor/rest/v1/tenants/{tenantId}/customers/{customerId}/wallets

WalletTypeId must be for a card mode.

You must populate one of the primaryPhysicalCardIndetifier fields. (packID , Pan or QrCode) to identify the card being issued to the customer.

Issue a secondary physical card

To issue a secondary physical card to an existing customer.

Prerequisites

A valid JWT

Have a physical card available for handover.

Will need the Wallet ID of the existing primary card you wish to link the secondary card to.

Step 1 – Create Card

Register and link the card using the Create Card

This refers to the following API in swagger:

POST /eclipse-conductor/rest/v1/tenants/{tenantId}/customers/{customerId}/cards

You must populate one of the primaryPhysicalCardIndetifier fields. (packID , Pan or QrCode)

Use the Wallet ID of the primary card (virtual card) wallet you wish to link the secondary card to.

Give Customer Eclipse API Access

Tenants may have use cases where their customers call the Eclipse API's directly and not via the tenant. This requires the customers to have an identity and password so that the customer front end (app, web etc) can obtain a valid JWT for calling Eclipse API's and only have access to their profile, wallets etc.

Prerequisites

A valid tenant JWT

Step 1 – Create Customer Identity and Password

Allow the customer to create their Identity and Password. This may be required for certain API authentication.

POST /eclipse-conductor/rest/v1/tenants/{tenantId}/customers/{customerId}/identities

Note that when passing a password for identity creation or updating, one can pass the new password in clear text or as a pre-hashed BCRYPT hash. The hash should ideally be a cost factor of 10 to 12. E.g. $2a$10$tMR2xhUpO0kEpJT9E7p2ZuiJaqxFVmpCJjYJG7H7ttstFq2ddjXfq

Search for a customer

You can search for a customer

Prerequisites

Valid Tenant JWT

Step 1 – Search customer

You can search for a customer using either :

Email, Last Name, ID number, Passport Number or phone number

GET /eclipse-conductor/rest/v1/tenants/{tenantId}/customers

Physical Card PIN Set

To set a PIN on a Physical Card.

Prerequisites

A valid JWT

A card issued to a customer and no PIN previously set.

CardID of the card being set.

The customer must have been thoroughly authenticated before allowing a PIN change request.

Step 1 – Set PIN

Allow the customer to set their card PIN

PUT/eclipse-conductor/rest/v1/tenants/{tenantId}/cards/{cardId}

Populate pin field

Note, that there may be additional out of band requirements to authenticate the customer, depending on the program configuration

Physical Card PIN Counter Reset

To reset a Card PIN counter if it has previously been exceeded.

Prerequisites

Valid JWT

CardID

The customer must have been thoroughly authenticated before allowing a PIN change request.

Step 1 – Reset PIN

Allow the tenant to reset their card PIN

PUT/eclipse-conductor/rest/v1/tenants/{tenantId}/cards/{cardId}

Pass a 0 in pinRetryCounter to reset the counter

Note, that there may be additional out of band requirements to authenticate the customer, depending on the program configuration

Physical Card PIN Change

Set a new PIN on a physical card

Prerequisites

Valid JWT

The customer must have been thoroughly authenticated before allowing a PIN change request.

CardID

Step 1 – Reset PIN

Change card PIN

PUT/eclipse-conductor/rest/v1/tenants/{tenantId}/cards/{cardId}

Pass new 4 digit PIN to set the new PIN

Note, that a customer must be authenticated before allowing a PIN change . Additional out of band requirements to authenticate the customer, depending on the program configuration

Transfer Funds Between Wallets

As a tenant, you are able to transfer funds between wallets. This is however dependent on the wallet type in use and regulatory environment at play. General access to transfer between any wallets in a tenant should be discussed beforehand with the Eclipse support team so that this can be turned on for the wallet type in question.

Prerequisites

Valid JWT with access to the source wallet

Sufficient funds available in the source wallet.

Sender Wallet ID and Receiver Wallet ID. A Wallet ID can be found by using the Customer ID and API
GET/eclipse-conductor/rest/v1/tenants/{tenantId}/customers/{customerId}/wallets

Step 1 – Wallet to Wallet Transfer

Perform the wallet to wallet transfer

POST/eclipse-conductor/rest/v1/tenants/{tenantId}/transfers

Due to Eclipse abstracting the underlying store of value and providing a consistent wallet API, it often needs to transfer between wallets which are on a different platform. E.g. a transfer from a digital wallet to a prepaid card wallet. Due to the nature of the SoV API's, transfers cannot always be done in an ACID (https://en.wikipedia.org/wiki/ACID) transaction. This is due to the fact that the debit leg and credit leg are on different isolated systems which do not support a 2-phase-commit protocol. To overcome this, Eclipse utilises the following approaches for transfers between different stores of value:

Transfer from Eclipse SoV to Non-Eclipse Sov

1. A transaction is started
2. The Eclipse wallet is debited
3. The non-Eclipse wallet is retrieved and verified its in a state to accept a credit
4. A retryable task is added into the Eclipse database to credit the Non-Eclipse wallet
5. The transaction is committed
6. A post-commit hook kicks off a process to try and retry the credit leg until it succeeds so that eventual consistency is achieved.

Typically eventual consistency is achieved within 1-2s depending on the latency of the destination SoV. If something goes wrong in the destination SoV between steps 3 and 6 then it will retry with a back-off algorithm forever until the destination is credited.

Transfer from Non-Eclipse SoV to Eclipse Sov

1. A transaction is started
2. The non-Eclipse wallet is debited
3. A post-rollback hook is enlisted to undo the debit (with a compensating credit) if the transaction is rolled back
4. The Eclipse wallet is credited
5. The transaction is committed

As can be seen in the above flows, when debiting Eclipse and crediting a non-Eclipse SoV, it is possible that the transfer is done and will be eventually consistent and yet if one checks the balance of the destination wallet immediately after the transfer, it may not show the new balance. This can be confusing for end users if a UI wants to show the new balance immediately after a transfer returns. To aid in this, the transfer API has a field called "replyPolicy" which can be used to make Eclipse only reply to the transfer after it was seen

that eventual consistency has been achieved. This alleviates the need for a front end to possibly poll and wait for the destination wallet to show the new balance.

The policy field has 2 options:

WHEN_GUARANTEED: Eclipse should reply when the result is guaranteed to happen and will eventually happen. I.e. Eclipse will reply after the transaction commit has happened.

WHEN_COMPLETE: Eclipse should only reply when the source and destination are fully consistent. I.e. after step 6 is successful in the scenario “Transfer from Eclipse SoV to Non-Eclipse Sov”

In all normal situations and by default, WHEN_GUARANTEED should be used. For most wallet transfers between wallets in the same store of value, WHEN_GUARANTEED and WHEN_COMPLETE are identical. WHEN_COMPLETE is only useful when transferring from a digital wallet within Eclipse to a store of value like a card wallet that sits outside of Eclipse.

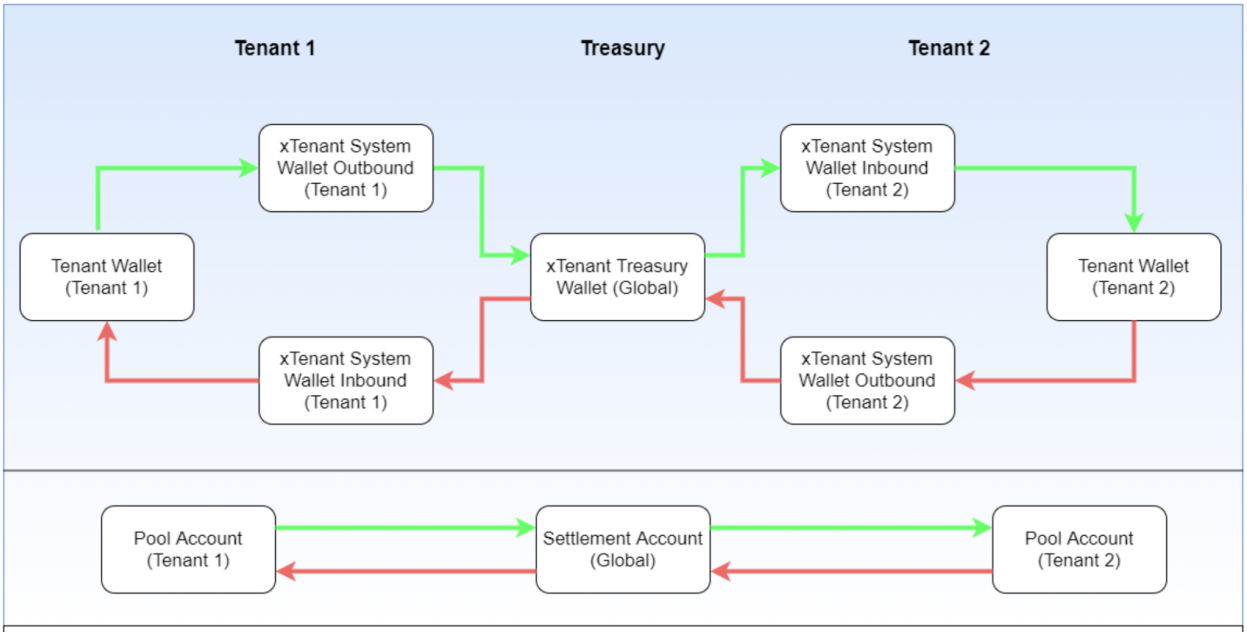
NB: If the destination wallet is not consistent within 5 seconds then Eclipse will reply as if WHEN_GUARANTEED was passed in, In order to prevent timeouts on the client. This is not considered a failure and the transfer API will still return HTTP 204 as the status code as the transfer is guaranteed to happen but has just not happened yet.

Cross Tenant Transfers

Tenants can transfer from their customer wallets to other tenant customer wallets so long as:

1. They have a wallet set up for outbound transfers to flow through. This is set as config value xTenantOutboundWalletId
2. The destination tenant wallet has the same currency as the source tenant wallet, as well as config value xTenantInboundWalletId.

The flow is as follows:



Digital Wallets

1. Tenant actions
   - Dr Tenant Wallet (Tenant 1) and Cr xTenant System Wallet Outbound (Tenant 1) (Destination Tenant and recipient set as a Reference)
2. System actions
   - Dr xTenant System Wallet Outbound (Tenant 1) and Cr xTenant Treasury Wallet (Global)
   - Dr Cr xTenant Treasury Wallet (Global) and Cr xTenant System Wallet Inbound (Tenant 2)
   - Dr xTenant System Wallet Inbound (Tenant 2) and Cr Tenant Wallet (Tenant 2)

Bank Accounts

1. Finance actions (end of day batch settlements)
   - All Credits to the xTenant Treasury Wallet (Global) are debited from the respective Pool Accounts and credited to the Settlement Account
   - All Debits to the xTenant Treasury Wallet (Global) are debited from the Settlement Account and credited to the respective Pool Accounts

A cross tenant transfer thus results in 8 ledger entries in total (4 DR/CR pairs). This is all done automatically by Eclipse. The transfer itself is done like any other transfer where the source and destination wallet details are provided.

Get Wallet Balance

A customer or tenant may request a wallet balance, subject to the program access rules. Note that a customer may have multiple wallets.

Prerequisites

Valid JWT with access to read the wallet

wallet ID

Step 1 – Wallet request

The below API will provide the details of a specific wallet:

GET/eclipse-conductor/rest/v1/tenants/{tenantId}/wallets/{walletId}

One can also get all the customers wallets

GET/eclipse-conductor/rest/v1/tenants/{tenantId}/customers/{customerId}/wallets

Get Wallet Transaction History (Statement)

A customer, organisation and Tenant may request a wallet balance history, subject to the program access rules. Note that a customer may have multiple wallets.

Prerequisites

Valid JWT with access to read the wallet

WalletId

Step 1 – Transactions request

Request a statement based on either a date range or a number of transactions

Note that card statements can return a maximum of 40 transactions

GET /eclipse-conductor/rest/v1/tenants/{tenantId}/wallets/{walletId}/transactions

Stop a Card

A customer's card may be stopped by the customer, organisation or tenant. It should be stopped immediately if the card is assumed to be lost or stolen.

Prerequisites

Valid JWT

Card ID

Step 1 – Stop Card

Request to stop the card

Change the card status to “LOST, STOLEN, DAMAGED, COUNTERFEIT ”

PUT/eclipse-conductor/rest/v1/tenants/{tenantId}/cards/{cardId}

Unstop a Card

A customer's card may be Unstopped by the customer, organisation or tenant. It can't be unstopped if the status is stolen or counterfeit.

Prerequisites

Valid JWT

Card ID

Step 1 – UnStop Card

Request to stop the card

Change the wallet status to "Active "

PUT/eclipse-conductor/rest/v1/tenants/{tenantId}/cards/{cardId}

Enable / Disable Card E-Commerce Transactions

A customers card may enable or disable e-commerce transactions (only if supported by the program)

Prerequisites

Valid JWT

Card ID

Step 1 – Enable / Disable Ecommerce on Card

Request to enable / disable .

Change the ecommerceTransactionEnabled card rule to True / False

PUT/eclipse-conductor/rest/v1/tenants/{tenantId}/cards/{cardId}

Enable / Disable Card International Transactions

A customer's card may enable or disable international transactions (only if supported by the program) .

Note: In South Africa, any international transactions must be either reported against either an Organisation or Customer, not a Tenant.

Prerequisites

Valid JWT

Card ID

Step 1 – Enable / Disable International

Request to enable / disable .

Change the internationalTransactionsEnabled card rule to True / False

PUT/eclipse-conductor/rest/v1/tenants/{tenantId}/cards/{cardId}

Do a Physical Card Replacement

To replace a card for a customer when required.

Prerequisites

If the card replacement is for a card that is still in active state, you should stop the active card e.g. replace a card that is soon to expire.

Valid JWT

Step 1 – Create Card

Register and link the card using the Create Card

This refers to the following API in swagger:

POST/eclipse-conductor/rest/v1/tenants/{tenantId}/customers/{customerId}/cards

You must populate one of the primaryPhysicalCardIndetifier fields. (packI , Pan or QR Code)

Use the Wallet ID of the primary card (virtual card) wallet you wish to replace.

Do a Virtual Card Replacement

To replace a virtual card for a customer when required

Prerequisites

If the card replacement is for a card that is still in active state, you should stop the active card. e.g. replace a card that is soon to expire.

Valid JWT

Step 1 – Create Card

Create the card using the Create Card

This refers to the following API in swagger:

POST/eclipse-conductor/rest/v1/tenants/{tenantId}/customers/{customerId}/cards

You must NOT populate one of the primaryPhysicalCardIndetifier fields.

Use the Wallet ID of the primary card (virtual card) wallet you wish to link the replacement card too

Retrieve Card Details for Customer (API)

Ability to retrieve full card details for a customer.

Prerequisites

A customer must be set up, with a card already created. Have a cardId ready.

Valid JWT

Step 1 – Create a CAP (Card Access Password)

Make API call to create a CAP (card access password) for the user (only needed once per customer)

POST /eclipse-conductor/rest/v1/tenants/{tenantId}/customers/{customerId}/identities

Here you post the identity as simply "CAP" (Card access password), no password field required.

{"identity":"CAP"}

Eclipse will generate a random 6 letter password and SMS it to the customers phone number on their profile with some generic wording like "Your password to access your virtual card details is xxxxxx. Please keep it safe as you will need it whenever you look up your card details". If the customer already has a CAP then we will set a new one and SMS the new one.

Step 2 - User enters CAP, then Retrieve Card

The user gets their card details from the app without needing a JWT but the app asks them to enter the password they received by SMS to access their card details:

GET
/eclipse-conductor/rest/v1/tenants/{tenantId}/cards/{cardId}?masked=false&cap=xxxxxx

NOTE: On Sandbox the SMS will not be sent, but a generic password ‘cap123’ can be used.

Retrieve Card Details for Customer (Captive Portal Flow)

Step 1 – Create Card

Prerequisites

Valid JWT of customer.

Users first need to create the card using a valid wallet type id.

POST

/eclipse-conductor/rest/v1/tenants/{tenantId}/customers/{customerId}/wallets

Step 2 – Send user to webview to set password for Card

Prerequisites

Customer/user must have to register a mobile number during the registration or provisioning process.

Valid cardId

Valid tenantId

Valid customerId

Once a user has the cardId, tenantId and customerId they can set the password to his/her card using the following request.

GET

<eclipse-base-url>/card-ui/?interactionType=createPin&userId=<user_id>&cardId=<card-id>&tenantId=<tenantId>

When a user hits the above URL in their browser, they will get an OTP on the registered mobile number. The user must enter a new password with that OTP number. If the OTP is valid, the password will be set, otherwise the user will get an error.

Step 3 – View Card

Prerequisites

tenantId

userId

cardId

password

Users can view his/her card using the following request.

GET

<eclipse-base-url>/card-ui/?interactionType=viewVirtualCard&tenantId=<tenant-id>&userId=<user_id>&cardId=<card-id>

Users need to enter the above URL in the browser then enter the valid password and the card data will be displayed.

Update customer information

To update a customers information

Prerequisites

Valid JWT

Customer ID

Step 1 – Update Customer information

Update info

This refers to the following API in swagger:

PUT /eclipse-conductor/rest/v1/tenants/{tenantId}/customers/{customerId}

Update customer address

To update a customers address

Prerequisites

Valid JWT

Pre-existing created addressId

Step 1 – Update the customer address

PUT /eclipse-conductor/rest/v1/tenants/{tenantId}/addresses/{addressId}

Update customer documentation

Update customer documentation

Prerequisites

Pre-existing created documentId

Valid JWT

Step 2 – Update the document

PUT/eclipse-conductor/rest/v1/tenants/{tenantId}/documents/{documentId}

Cancel Customers Wallets & Cards

If a customer account needs to be cancelled and all associated cards cancelled (if any)

Prerequisites

The wallets available balance should be zero.

Valid JWT

Step 1 – Delete the wallet

DELETE/eclipse-conductor/rest/v1/tenants/{tenantId}/wallets/{walletId}

Customer KYC Validation and Update

If required, you can validate and update a customer's KYC documents and KYC status.

Prerequisites

Customer Documents to be created and uploaded.

customerId

Valid JWT

Step 1 – Update profile, addresses, documents

PUT/DELETE the customers profile, documents and addresses with the necessary changes

Step 2 – Validate Customer using Ratify

Run a KYC algorithm on the customer profile and update the KYC status

POST/eclipse-conductor/rest/v1/tenants/{tenantId}/customers/{customerId}/ratify

This will return a detailed status of the checks passed/failed

Create a QR Code linked to a Wallet

It is possible to create a QR Code for acquiring purposes, if enabled for the wallet type.

Prerequisites

Pre-created Wallet

walletId

Valid JWT

Step 1 – Create QR Code linked to the wallet

Create a QR Code which is linked to the wallet.

POST/eclipse-conductor/rest/v1/tenants/{tenantId}/wallets/{walletId}/qr-codes

Pull QR Code details

It receive QR Code and wallet details

Prerequisites

QR Code

Valid JWT

Step 1 – Get QR Code details

GET/eclipse-conductor/rest/v1/tenants/{tenantId}/wallets/qr-codes/{code}

Wallet Top-Up

Wallets are most useful when they have money in them. This money can come from incoming payments, or the customer topping them up directly with their own money - be it cash, card or EFT. Eclipse has multiple integrations with partners to simplify and provide numerous top up mechanisms for a tenants customers. Eclipse integrates into multiple payment gateways and retailers and provides a single API to tenants. As a Tenant or Organisation, if you use your own internal Top up methods, then a Wallet to Wallet transfer should be used.

Prerequisites

walletId

Valid JWT

Step 1 – Initiate topup process

The overall process depends on the type of top up. This could result in an iframe being presented to the customer to enter card details for topping up, or could return a token for topping up at a retail store (e.g. PnP).

POST /eclipse-conductor/rest/v1/tenants/{tenantId}/wallets/{walletId}/topups

Alternatively EFT into Pool account

A customer can also get funds directly into their wallets by way of doing a standard EFT into the tenant Pool bank account with the wallet's Unique Reference('friendlyId') as the EFT reference.

Wallet Withdrawals - Non Card

A time will come where a customer wants to move money off the Eclipse ecosystem into a bank or as cash. Eclipse facilitates this process through numerous integrations for withdrawing cash and electronically moving money into traditional bank accounts. The numerous integrations are consolidated into a single simple API for initiating a withdrawal. Strict rules are enforced based on the authenticated user doing the withdrawal and their relationship to the wallet being debited. This is based on who the wallet is owned by, or in the case of an organisation, what position the caller holds in that organisation. There are also rules based on the wallet type which determine if the tenant system API calls have authorisation to perform withdrawals on their customers wallets

Prerequisites

walletId

Valid JWT

Step 1 – Initiate withdrawal process

The overall process depends on the type of withdrawal. This could return a token for withdrawal at a retail store (e.g. PnP) or kick off an EFT.

POST/eclipse-conductor/rest/v1/tenants/{tenantId}/wallets/{walletId}/withdrawals

Create a Card to Card Money Send request (Not available yet)

To be updated when available.

Organisation Use Cases

Create an Organisation

To create an organisation

Prerequisites

KYB documentation available.

Search organisation to look for potential duplicates within the Tenant.

Need organisations information (Name, Trading Name, Tax Number Registration Number, Address etc)

Valid JWT

Step 1 – Create Organization

Create the Organisation

This refers to the following API in swagger:

POST /eclipse-conductor/rest/v1/tenants/{tenantId}/organisations

Step 2 – Add Address Details

Add address details for the organisation now that you have an Organisation ID

This refers to the following API in swagger:

POST/eclipse-conductor/rest/v1/tenants/{tenantId}/organisations/{organisationId}/addresses

Step 3 – Add Supporting Documentation

Add Organisation supporting documentation.

This refers to the following API in swagger:

POST/eclipse-conductor/rest/v1/tenants/{tenantId}/organisations/{organisationId}/documents

Step 4 – Kick off Ratify

POST /eclipse-conductor/rest/v1/tenants/{tenantId}/organisations/{organisationId}/ratify

This will kick off the various checks and algorithms in Eclipse to analyse the organisation profile, address/s and attached documents to update the organisations KYB status. This call will return detailed results of the KYB process such as what passed and what failed in the various tests. The organisation profile can be updated along with addresses and documents and ratified again until the required KYB status for the wallet type is obtained. The wallet type determines what level of KYB is required for a customer/organisation to own that type of wallet.

Step 5 – Create Wallet

Create a wallet which will be used to top up customer wallets

This refers to the following API in swagger:

POST/eclipse-conductor/rest/v1/tenants/{tenantId}/organisations/{organisationId}/wallets

Add Positions to Organisation

To link a customer as having a position in an organisation

Prerequisites

Create the Customer and have the customerId

Organisation ID

Valid JWT

Step 1 – Create Customer Position within the Organisation

Post the position of the customer

This refers to the following API in swagger:

POST/eclipse-conductor/rest/v1/tenants/{tenantId}/organisations/{organisationId}/positions

Register a new customer as part of an organisation and issue a virtual card linked to an Organisation wallet

In order to sign up a new customer and issue a virtual card, first create the customer along with the required customer information and then create the wallet under an organisation passing the customerId.

(With the creation of the Wallet, the card is automatically created)

Follow steps 1 -3 of “Register a new customer and issue a virtual card”

Step 4 – Create Wallet under Organisation

Create a wallet which will automatically be created and link the virtual card.

This refers to the following API in swagger:

POST /eclipse-conductor/rest/v1/tenants/{tenantId}/organisations/{organisationId}/wallets

Do **not** populate the primaryPhysicalCardIndetifier fields.

Set the customerId to the customer who will be the first cardholder on the wallet.

Organisation KYC Validation and Update

If required, you can validate and update an organisation's KYB documents and KYB status.

Prerequisites

Organisation Documents to be created and uploaded.

organisationId

Valid JWT

Step 1 – Update profile, addresses, documents

PUT/DELETE the organisation's profile, documents and addresses with the necessary changes

Step 2 – Validate Organisation using Ratify

Run a KYB algorithm on the organisation profile and update the KYB status

POST/eclipse-conductor/rest/v1/tenants/{tenantId}/organisations/{organisationId}/ratify

This will return a detailed status of the checks passed/failed

Create an Organisation Wallet

To create an Organisation Wallet.

Prerequisites

Pre-existing organisationId

Valid JWT

Step 1 – Create an organisation wallet

POST/eclipse-conductor/rest/v1/tenants/{tenantId}/organisations/{organisationId}/wallets

Request an Organisation Wallet Balance

An organisation may request a wallet balance, subject to the program access rules. Note that a organisation may have multiple wallets

Prerequisites

Pre-existing organisation wallet creation and balance details:

tenantId

walletId

Valid JWT

Step 1 – Request organisation wallet balance

Perform a single wallet balance request.

GET/eclipse-conductor/rest/v1/tenants/{tenantId}/wallets/{walletId}

You can also get multiple wallet balances using

GET/eclipse-conductor/rest/v1/tenants/{tenantId}/organisations/{organisationId}/wallets

Get Wallet Transaction History (Statement)

Same as for a customer's wallet

Update Organisation information

To update an organisation information

Prerequisites

Pre-existing organisation information:

tenantId

organisationId

Valid JWT

Step 1 – Update organization Information

PUT /eclipse-conductor/rest/v1/tenants/{tenantId}/organisations/{organisationId}

Update Organisation address

Prerequisites

Pre-existing organisation address:

tenantId

AddressID
GET/eclipse-conductor/rest/v1/tenants/{tenantId}/organisations/{organisationId}/addresses)

Valid JWT

Step 1 – Update organization address

PUT/eclipse-conductor/rest/v1/tenants/{tenantId}/addresses/{addressId}

Update Organisation documentation

Prerequisites

Pre-existing organisation documentation

tenantId

documentId

Valid JWT

Step 1 – Update organization documentation

PUT/eclipse-conductor/rest/v1/tenants/{tenantId}/documents/{documentId}

Payments Use Cases

Payments allow tenants to initiate or process a payment from a store of value - sometimes one on Eclipse and sometimes a card or store of value outside of Eclipse. In many ways, there is an overlap between payments and topups in that a topup can often be seen as a payment to one's own wallet. As payments can result in funds going to non-Eclipse wallets, they are a superset of topups. I.e. topups always result in funds ending up in an Eclipse Wallet while payments can result in funds going to an external bank/wallet/account. When in doubt, use topups APIs when the customer carrying out the transaction is going to be the recipient of the funds into their Eclipse wallet, and payments where the person or API doing the payment is not going to be the one sending and receiving the funds. The payments API can be used at the customer level or directly at the tenant level. Using it at the customer level allows Eclipse to deduce the payer information while at the tenant level the API requires customer information to be passed in. From there on both API's are identical.

Creating a payment involves 2 core pieces of data. The first is the combination of paymentData and type. These determine where the money is going to. The second is the payment mechanism (i.e. where the money is coming from) which can be captured in fields like walletId, externalWalletId & externalWalletType, paymentCardData, and finally, cardOnFileId. Eclipse uses this info to decide how to get the funds from where it’s coming from to where it’s going to.

Due to the numerous payment mechanisms and payment flows, the payments API supports both single and multi-step payments depending on the type of payment and payment mechanism.

- Multi step payments involve a POST and then one or more PUT operations until the payment is fully processed. This is needed where the POST initiates a payment and then additional data is required to complete it.
- Single step payments involve a POST with all required information and the payment is done without further PUT’s required.

As each payment type can involve different fields, they are documented separately but all use the same endpoint and JSON structure. Just some use more or less fields than others and may need more or less steps to achieve a final outcome.

Also note that payments are often associated with the rendering or provisioning of a value added service once the payment is made. This is orchestrated by Eclipse or by the VAS partner depending on what is being purchased. API callers can assume that Eclipse is taking care of ensuring that a payment cannot be made without the VAS being delivered and vice versa.

Payment Type: ZA_QRCODE

This payment type should be used when paying a Masterpass QRCode in South Africa.

It can be a single or multi-step process.

To initiate a transaction for a specific customer:

POST/eclipse-conductor/rest/v1/tenants/{tenantId}/customers/{customerId}/payments

Or if not for a specific customer:

POST/eclipse-conductor/rest/v1/tenants/{tenantId}/payments

The only difference between the two endpoints is that the customer one will use the customer's details where a payment gateway needs customer information, whereas the second one requires the customer info to be passed in. Hence the second one having an additional field “phone” to indicate the phone number of the customer making the payment. For the second one, phone number is required for Masterpass and should be in an international format starting with country code (e.g. 27831234567).

Other than phone, the following are required fields:

externalUniqueId: A unique GUID to identity this payment and prevent unintended duplicate payments

paymentData: The decoded ten digit masterpass QRCode e.g. 9876543210

type: ZA_QRCODE

The response to the POST will include information about the QRCode (amount and whether additional fields are required). The status will be PENDING if the POST had sufficient data to send to Masterpass and BUILDING if more data is still required.

Indicative response:


```
{"paymentId":3034,"status":"BUILDING","amount":15.00,"description":"Ukheshe Test 2021  - Me :
Test QR","merchantName":"Ukheshe Test 2021  -
Me","currency":"ZAR","partialPaymentAllowed":false,"acceptedCardSchemes":["MASTER","VISA","MAESTR
O","AMEX","DINERS"],"acceptedPaymentMechanisms":["AMT","SECURE_CODE","SOV"]}
```


At this point, if the status is BUILDING then the next step is a PUT to update the payment further on URL /eclipse-conductor/rest/v1/tenants/{tenantId}/payments/{paymentId}

The data in the put should include how the QR must be paid and any additional fields needed for the QR.

There are 5 supported means of specifying the source store of value for paying a Masterpass QR:

1. Providing a card on file Id for a previously added card on file. This must be a CoF belonging to the caller (JWT).
2. For PCI-DSS compliant tenants, the card data can be provided directly in the paymentCardData field.
3. For tenants with their own store of value and prior integration, they can pass their own externalWalletId and externalWalletType (e.g. ABC12345 and MOBI_CHOICE)
4. For tenants with a lodge card registered to do card rails payments on behalf of their closed loop Eclipse digital wallet, the Eclipse walletId can be provided
5. Providing none of the above will result in a completionUrl being returned where the customer can capture card data in a light box (iFrame) and complete 3DS and the payment.

The value of the paymentSecurityData field depends on the store of value paying the QRCode. If the tenant is the card issuer (bank) they can provide the 3DS fields such as cavv so as to skip 3DS. External stores of value may pass a token as additional authentication (TBD with tenant).

callbackUrl: Where Eclipse will post the final result when Masterpass has completed the transaction (optional)

landingUrl: Where to navigate the customer at the end of a lightbox process if the customer has to complete card details / 3DS

Amount: Must be populated with the final total amount the customer will pay. Required

Payment mechanism is not needed as it can be deducted from the other data.

Example PUT body specifying a card on file to pay with:

{"landingUrl":"http://www.blanksite.com/","cardOnFileId":"e58c8bcf-f085-436f-90e2-4916f4b90efc","amount":15.00}

Indicative response:

{"paymentId":3034,"externalUniqueId":"1fe9338f-06ed-4654-afdf-07b7c00c3c3e","status":"PENDING","paymentType":"CARD","description":"Ukheshe Test 2021 - Me : Test

QR","amount":15.00,"completionUrl":"https://ukheshe.live/t/7amar","created":"2021-06-08T10:12:56.000Z","lastModified":"2021-06-08T10:12:56.767Z","paymentData":"9455383854"}

At this point the customer can be navigated to the completionUrl to complete 3DS after which they will be sent to the landingUrl

When the payment completes (failed or success), a callback will be done to callbackUrl with the result of the payment in the same format as if a GET was done on the paymentId.

If a Masterpass payment has a status of SUCCESSFUL then the card/wallet has been debited and the merchant has been notified of the successful transaction and the customer can get their goods/services.

Please refer to this section for sample qr's and use cases

Payment Type: GLOBAL_PAYMENT_LINK

Payment links can be created as a simple means of requesting a payment which will clear into a specific wallet. Once the link is generated, it can be shared with the payer, who can open the link in a browser and decide how to make the payment by choosing one of many payment mechanisms depending on the tenant configuration. These may include card, closed loop wallet providers, cash deposit instructions, EFT instructions etc.

To generate a payment link, pass the currency, amount, type and a GUID in externalUniqueId. Also include an additional field called destinationWalletId with the value being the walletId that must be credited when the payment is made.

Example payload:

{"externalUniqueId":"193b88b5-a448-418a-84ed-0c8f15c95c5b","type":"GLOBAL_PAYMENT_LINK","amount":123,"currency":"ZAR", "callbackUrl":"https://letmeknowwhendone.com/abc","additionalFields":[{"id":"destinationWalletId","value":"12345"}]}

Example response:

{"paymentId":3117,"status":"PENDING","amount":123.000000,"currency":"ZAR","additionalFields":[{"id":"destinationWalletId","value":"12345"}],"completionUrl":"https://ukheshe.live/t/e94EE"}

The customer can then be sent the completionUrl or it opened for them in their browser and they can complete the payment. When the payment completes, a callback is done to the callbackUrl with the result. A payment link can only be paid once.

The tinyUrl returned in completionUrl is only valid for 24h. If a longer duration is needed then the URL can be fetched to get the final destination and used instead. The actual underlying payment link never expires.

Payment Type: GLOBAL_VAS

TBC

Payment Reversals

Once a payment is successful, the recipient can choose to reverse the payment back to the payer. This functionality depends on the payment type as some payment methods are impractical to reverse or simply do not support this technically.

In order to reverse a payment, the PUT endpoint can be used to change the status of the payment to REVERSED:

If the gateway supports this, then the API will return and indicate that the payment status is now REVERSED. Eclipse will take care of ensuring the funds are debited from the payment destination wallet and credited to the customers payment source.

Payment Status State Diagram

Payments have the following potential statuses:

BUILDING: Being built up and does not have sufficient information to submit downstream and become PENDING

PENDING: Being processed by the downstream gateway

SUCCESSFUL: Successfully completed

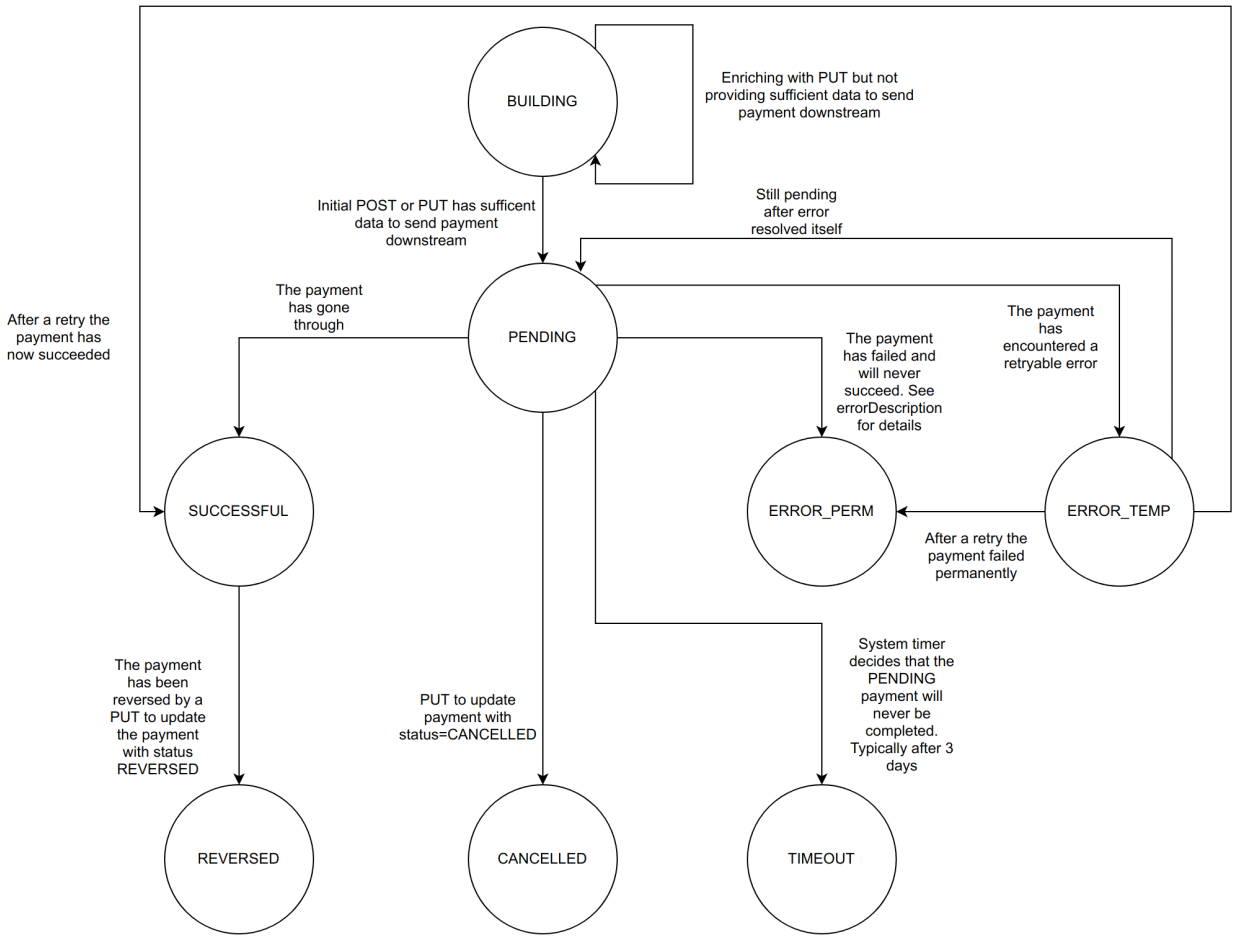
ERROR_PERM: Failed and won’t ever succeed,

ERROR_TEMP: Currently having an error but it's being retried and may succeed or permanently fail. (e.g. the downstream provided got a system error processing our last request for a status update)

TIMEOUT: Failed as the process was not finalised in time

CANCELLED: Cancelled while in PENDING status

REVERSED: Reversed after being SUCCESSFUL


BUILDING
Enriching with PUT but not providing sufficient data to send payment downstream
Initial POST or PUT has sufficent data to send payment downstream
Still pending after error resolved itself
After a retry the payment has now succeeded
The payment has gone through
PENDING
The payment has failed and will never succeed. See errorDescription for details
The payment has encountered a retryable error
SUCCESSFUL
ERROR_PERM
ERROR_TEMP
After a retry the payment failed permanently
The payment has been reversed by a PUT to update the payment with status REVERSED
PUT to update payment with status=CANCELLED
System timer decides that the PENDING payment will never be completed. Typically after 3 days
REVERSED
CANCELLED
TIMEOUT

Global Integrator Use Cases

Global integrators utilise a set of API's which are not specific to any one tenant. These integrators are a special type of tenant on the platform that does not have any customers/organisations but rather is allowed to send money from their own wallet/s to a wallet in any other tenant. This gives them the ability to disburse funds to any tenant in a secure, controlled fashion - such as for payouts, payroll, loyalty gifts etc.

The tenantId and walletId/s belonging to the global integrator will be communicated to the integrator and can be queried using the standard tenant API's. The integrator's wallet can be funded by EFT and other bulk mechanisms organised by prior arrangement. The most relevant API's for these are included below for brevity:

There are 5 specific use-cases for global integrators:

Authentication

This is the same authentication mechanism as for any tenant. As external integrators have wallets with a large balance, their identity will be created with PKI so that there is mutual authentication before a JWT will be issued.

Follow this section on how to get a JWT

Customer/Wallet Lookup

Customer lookup is used in order to get the unique wallet Id on Eclipse based on information that a customer could supply to the global integrator. This allows the integrator to find the correct wallet on Eclipse if the customer happens to have multiple wallets or is not aware of the walletId of their wallet. If the customer does not have a wallet on Eclipse then the integrator can instruct them on how to sign up on one of the Eclipse tenants channels (App, Web, WhatsApp, USSD etc). Once the customer has registered on a tenant, they will be included in the results of this lookup.

As the data on Eclipse is private, this API will not disclose personal information about the customer but will provide enough for the integrator to identify if the wallet is the one they want to transfer to. At least two of the search criteria should match in order for a record to be returned in the list. The result will return a limit of 100 records. This is to ensure the privacy of Eclipse customers and prevent generic searching for wallets. This API is rate

limited to prevent bulk calling so API callers should store the unique walletId when they find the wallet they are looking for as opposed to searching for it during each payroll run.

Obfuscated fields will contain “*” in various fields to hide the full value but provide enough information for the owner to know it is their data.

Request:

GET /eclipse-conductor/rest/v1/global/wallets Search for a wallet irrespective of tenant

Returns information of a wallet without disclosing personal details. At least 2 of the supplied search fields must match the customer data in order for a record to be returned. This is to ensure the privacy of Eclipse customers and prevent generic searching for wallets. This API is rate limited to prevent bulk calling so callers should store the unique walletId when they find the wallet they are looking for as opposed to searching for it during a payroll run.

Parameters | Try it out

Name	Description
lastName string (query)	Customers last (family) name of max length 50 lastName - Customers last (family) name of r
nationalIdentityNumber string (query)	Max length 20 nationalIdentityNumber - Max length 20
passportNumber string (query)	Max length 20 passportNumber - Max length 20
phone string (query)	Either with or without the country code. E.g. 0821112222 or 27821112222 phone - Either with or without the country coc
walletFriendlyId string (query)	Will be length 8 if provided walletFriendlyId - Will be length 8 if provided
walletId integer($int64) (query)	walletId

Result:

Example Value | Schema

```
[
  {
    "lastName": "string",
    "nationalIdentityNumber": "string",
    "passportNumber": "string",
    "phone1": "string",
    "phone2": "string",
    "walletFriendlyId": "string",
    "walletId": 0
  }
]
```

Where:

lastName	string Obfuscated last name. Only present if the owner has this field on their profile
nationalIdentityNumber	string Obfuscated national identity number of the wallet holder. Only present if the owner has this field on their profile
passportNumber	string Obfuscated passport number of the wallet holder. Only present if the owner has this field on their profile
phone1	string Obfuscated phone number 1 of the wallet holder. Only present if the owner has this field on their profile
phone2	string Obfuscated phone number 2 of the wallet holder. Only present if the owner has this field on their profile
walletFriendlyId	string A unique string reference for a wallet used for deposit references and other transactions
walletId	integer($int64) A unique system generated numeric identifier for the wallet. This is the required identifier for doing transfers to the wallet in field toWalletId

Wallet Transfer to Customer

A wallet transfer can be initiated from the integrator's wallet to a wallet on any other tenant. The destination walletId could be provided by the end customer, or searched for using the [customer wallet/lookup API](). Once the funds are in the customers wallet, they can use their wallet with the providers channels to withdraw, transfer etc.

Request:

POST /eclipse-conductor/rest/v1/tenants/{tenantId}/wallets/transfers Do a wallet transfer

Parameters | Try it out

Name	Description
tenantId * required integer($int64) (path)	tenantId

Request body | application/json

Example Value | Schema

```
{
  "amount": 0,
  "description": "string",
  "externalId": "string",
  "externalUniqueId": "string",
  "fromWalletId": 0,
  "sessionId": "string",
  "toWalletId": 0
}
```

Where:

amount	number *minimum: 1* **Amount to transfer**
description	string *pattern: \S* *nullable: false* **Description for the statement/s**
externalId	string *maxLength: 50* *pattern: \S* *nullable: false* **External identifier for the transfer which can be used for reconcilliation,. Need not be unique**
externalUniqueId*	string *maxLength: 50* *minLength: 6* *nullable: false* **Unique identifier for the transfer (idempotency key). Duplicates are rejected to ensure double transactions are impossible**
fromWalletId	integer($int64) *minimum: 1* **The wallet to debit**
sessionId	string *maxLength: 200* **Optional: Used for 2-step reserve/transfer transactions. The sessionId of a reservation in order to make this transfer first remove the reservation and then do the transfer atomically**
toWalletId	integer($int64) *minimum: 1* **The wallet to credit**

The result is immediate and atomic. If an HTTP result code >= 200 < 300 is returned then the transfer succeeded. Any other result indicates a failure and the transaction was not done. If unsure, one could call the transaction query API below to see if there is a transaction with the supplied externalUniqueId. It is recommended that externalUniqueId is a GUID generated by the caller.

Failures return a JSON body with details as per section on error handling.

Direct Withdrawal

Direct withdrawal via ATM/Cash allows the integrator to send a withdrawal token for an ATM or retailer to a customer.

The integrator provides the customer phone number and Eclipse SMS's the withdrawal token & instructions to the customer. Eclipse verifies that the destination phone number does not receive more than R24 000 per month in withdrawal tokens.

In order to kick off a withdrawal token SMS, one can follow the standard withdrawal API call passing the integrators walletId as the source wallet for the withdrawal along with the customers mobile phone number:

Request:

POST /eclipse-conductor/rest/v1/tenants/{tenantId}/wallets/{walletId}/withdrawals Initiate a new withdrawal

The withdrawal type determines exactly what fields are required

Parameters

Try it out

Name	Description
tenantId * required integer($int64) (path)	tenantId
walletId * required integer($int64) (path)	walletId

Request body

application/json

Example Value | Schema

```
{
  "accountName": "string",
  "accountNumber": "string",
  "amount": 0,
  "bank": "string",
  "branchCode": "string",
  "callbackUrl": "string",
  "deliverToPhone": "string",
  "description": "string",
  "externalUniqueId": "string",
  "reference": "string",
  "type": "ZA_NEDBANK_EFT"
}
```

Where:

accountName	string maxLength: 50 minLength: 3 Account name to EFT to if applicable
accountNumber	string maxLength: 20 minLength: 10 Account number to EFT to if applicable
amount*	number minimum: 1 nullable: false The amount in the currency of the wallet
bank	string maxLength: 50 minLength: 3 Bank to EFT to if applicable
branchCode	string maxLength: 20 minLength: 3 Branch code to EFT to if applicable
callbackUrl	string maxLength: 2000 minLength: 10 The URL Eclipse must post the EclipseWalletWithdrawal to when the withdrawal is finalised (either successful or error). Optional as one can always do a GET to get the result instead of using callbacks. This will be a system-system call
deliverToPhone	string Phone number to SMS withdrawal details to. If not provided then the withdrawal token details will be returned in the API and count towards the limits of the customer owning the wallet
description	string Optional description or reference
externalUniqueId*	string maxLength: 50 minLength: 6 nullable: false Unique identifier for the withdrawal Duplicates are rejected to ensure double transactions are impossible
reference	string maxLength: 50 minLength: 3 Reference to EFT to if applicable
type*	string nullable: false The type of the withdrawal Enum: > Array [6]

For a cash withdrawal, one would pass an amount, callbackUrl (optional), description, externalUniqueId and type.

Types are as follows:

WalletWithdrawalType string

Currently supported withdrawal types. ZA_PNP_CASH:Cash out at any Pick n Pay in South Africa, ZA_PNP_TENDER:Pay for goods at any Pick n Pay in South Africa, ZA_NEDBANK_EFT:EFT to any South African bank account via Nedbank CPS, ZA_PAYACCSYS_ATM:Withdraw cash at a Standard bank ATM with a token/pin, ZA_SHOPRITE_CASH:Cash out at any Checkers/Shoprite in South Africa, ZA_PAYCORP_ATM:Withdraw at any CashExpress ATM

Enum:

[ZA_NEDBANK_EFT, ZA_PAYACCSYS_ATM, ZA_PAYCORP_ATM, ZA_PNP_CASH, ZA_PNP_TENDER, ZA_SHOPRITE_CASH]

Result:

Example Value | Schema

```
{
  "amount": 0,
  "currency": "string",
  "errorDescription": "string",
  "expires": "2021-02-04T13:46:18.153Z",
  "externalUniqueId": "string",
  "finalAmount": 0,
  "gateway": "string",
  "status": "CANCELLED",
  "subType": "string",
  "token": "string",
  "type": "string",
  "walletId": 0,
  "withdrawalId": 0
}
```

Where:

amount	number
currency	string
errorDescription	string This will only be populated if there was an error
expires	string($date-time) When the token expires if applicable
externalUniqueId	string
finalAmount	number
gateway	string
status	string As the callback will only trigger when the the withdrawal is in a final status, the possibilities are SUCCESSFUL, ERROR, TIMEOUT, CANCELLED Enum: [CANCELLED, ERROR, PENDING, REVERSED, SUCCESSFUL, TIMEOUT, VALIDATED]
subType	string
token	string The token in case the withdrawal is via a token/identifier given at time of withdrawing
type	string
walletId	integer($int64)
withdrawalId	integer($int64)

If no deliverToPhone was provided, then the token along with instructions for the specific withdrawal type can be given to anyone who can then use the token at the participating withdrawal channel (e.g. specific ATM brand, PnP etc) to complete the withdrawal.

If deliverToPhone was provided then the phone will receive instructions on how to withdraw the cash.

A callback will the withdrawal result will be done when the withdrawal is either successful or has an error or timed out. If it fails, the funds will remain in the integrators wallet.

The callback will contain the same data as the first response to the call, but now with the final status. One can also call the withdrawal query API at any time to get the status of a withdrawal:

GET `/eclipse-conductor/rest/v1/tenants/{tenantId}/wallets/{walletId}/withdrawals/{withdrawalId}` Get a withdrawal by Id

The Id is the withdrawal id field.

Parameters

Name	Description
tenantId * required `integer($int64)` *(path)*	tenantId
walletId * required `integer($int64)` *(path)*	walletId
withdrawalId * required `integer($int64)` *(path)*	withdrawalId

The result is the same as what would be sent in the callback:

```
amount                number
currency              string
errorDescription      string

                      This will only be populated if there was an error

expires               string($date-time)

                      When the token expires if applicable

externalUniqueId      string
finalAmount           number
gateway               string
status                string

                      As the callback will only trigger when the the withdrawal is in a final status, the possibilities are SUCCESSFUL, ERROR, TIMEOUT, CANCELLED

                      Enum:

                       > Array [ 7 ]
subType               string
token                 string

                      The token in case the withdrawal is via a token/identifier given at time of withdrawing

type                  string
walletId              integer($int64)
withdrawalId          integer($int64)
```

Some more details on the status values:

SUCCESSFUL - The withdrawal has actually been done and the recipient now has the cash

ERROR - Something went wrong and the withdrawal will not be possible. The errorDescription would explain what went wrong.

TIMEOUT - This applies to tokens which have an expiry and the expiry has been reached prior to the token being used. The money is available in the integrators wallet.

CANCELLED - A token was deleted prior to it being used. The money is available in the integrators wallet.

Integrator Wallet Enquiry

Global integrators will have one or more wallets in their own tenant which will be set up for them based on their needs. These will be used for disbursing funds to wallets in other tenants. The tenantId and walletId will be communicated to the integrator so that the standard wallet enquiry API can be called.

Request:

GET /eclipse-conductor/rest/v1/tenants/{tenantId}/wallets/{walletId} Get wallet by walletId.

Get a wallet when the walletId is known

Parameters | Try it out

Name	Description
tenantId * required integer($int64) (path)	tenantId
walletId * required integer($int64) (path)	walletId

Result:

Example Value | Schema


```
{
  "availableBalance": 0,
  "created": "2021-02-04T13:25:48.836Z",
  "currency": "string",
  "currentBalance": 0,
  "customerId": 0,
  "description": "string",
  "externalUniqueId": "string",
  "friendlyId": "string",
  "name": "string",
  "organisationId": 0,
  "reservations": 0,
  "status": "ACTIVE",
  "walletId": 0,
  "walletTypeId": 0
}
```


Where:

availableBalance	number The amount available for spending
created	string($date-time) ISO 8601 date/time of when the record was first created
currency*	string *nullable: false* The currency of the wallet - e.g. USD, ZAR, NGN, TSH
currentBalance	number The balance in the wallet in the currency of the wallet
customerId	integer($int64) The customer who owns this wallet. If not provided then the wallet must be owned by an organisation and hence organisationId should be provided
description	string *maxLength: 200* *minLength: 0* Wallet Description
externalUniqueId	string A globally unique identifier set by the tenant for this wallet. Useful for correlating between the tenants systems and Eclipse
friendlyId	string A unique reference for a wallet used for deposits and other transactions
name	string *maxLength: 50* *minLength: 3* Name of the Wallet
organisationId	integer($int64) The organisation who owns this wallet. If not provided then the wallet must be owned by a customer and hence customerId should be provided
reservations	number The reservations placed on the wallet (uncommitted transactions)
status*	string *nullable: false* The status of the wallet. Impacts if it can currently be used Enum: [ACTIVE, LOCKED]
walletId	integer($int64) *minimum: 1* A unique system generated numeric identifier for the wallet
walletTypeId	integer($int32) *minimum: 1* The type of wallet. This controls the behaviour and rules of the wallet

Transfer Transaction Query

To query an individual transfer, one can use the reconciliation query and pass the externalUniqueId that was used when the transfer was done. See the Reconciliation Query below for details.

Reconciliation Query

Global integrators will have one or more wallets in their own tenant which will be set up for them based on their needs. These will be used for disbursing funds to wallets in other tenants. The tenantId and walletId will be communicated to the integrator so that the standard wallet transactions API can be called:

Request:

GET /eclipse-conductor/rest/v1/tenants/{tenantId}/wallets/{walletId}/transactions Get wallets transactions.

Retrieves the transaction history for a wallet

Parameters

Try it out

Name	Description
dateFromIncl string (query)	fromDate, the date from where we want to get the transactions inclusive. dateFromIncl - fromDate, the date from where
dateToExcl string (query)	toDate, the date upto when we want to get the transactions exclusive dateToExcl - toDate, the date upto when we v
externalUniqueId string (query)	The externalUniqueId used when the transaction was initiated. Other parameters other than walletId are ignored if this is passed externalUniqueId - The externalUniqueId use
limit integer($int32) (query)	*Default value* : 50 50
offset integer($int32) (query)	*Default value* : 0 0
tenantId * required integer($int64) (path)	tenantId
walletId * required integer($int64) (path)	walletId

Result:

Example Value | Schema

```
[
  {
    "amount": 0,
    "authorisationCode": "string",
    "balance": 0,
    "currency": "string",
    "date": "2021-02-04T13:13:24.997Z",
    "description": "string",
    "externalId": "string",
    "externalUniqueId": "string",
    "fee": 0,
    "otherWalletId": 0,
    "transactionId": "string",
    "type": "string",
    "walletId": 0
  }
]
```

Where:

amount	number Amount. Negative for debits, positive for credits.
authorisationCode	string Auth code for card transactions
balance	number Closing balance in the currency of the wallet.
currency	string Currency of the transaction
date	string($date-time) ISO 8601 of the transaction
description	string Transaction description
externalId	string External identifier for the transaction which can be used for reconcilliation. Need not be unique
externalUniqueId	string The externally provided unique identifier for the transaction
fee	number Fee. The additional fee charged as part of the transaction
otherWalletId	integer($int64) If this was a transfer then this field will contain the other walletId which was debited/credited
transactionId*	string *nullable: false* Unique identifier for the transaction generated by the wallet/card system
type	string Transaction type
walletId*	integer($int64) *minimum: 1* *nullable: false* The walletId of the transaction

Exposing A Tenants Store of Value as a wallet type in Eclipse

Eclipse API's for topups, payments, withdrawals, remittance, transfers etc typically result in a debit/credit pair across 2 wallets. Wallets in turn have a wallet type that determines the rules and underlying implementation of the wallets store of value. Tenants who already have a store of value and ledger and want to compliment it with Eclipse functionality have 2 options:

1. The tenant controls the synchronization between Eclipse and their store of value by using webhooks, the event stream and coordinating calls between their SoV and Eclipse as part of a distributed transaction and keeping their store of value eventually consistent
2. The tenant always calls Eclipse for all wallet functionality and Eclipse in turn calls back to the tenant by having a wallet type whose implementation is the tenant's SoV.

Due to various scenarios like long-running payments, card transactions, ATM withdrawals etc, there are many cases where a tenant's wallet needs to be debited/credited without the tenant being the one that initiates the transaction at that point in time. For example an ATM withdrawal can happen 2 days after the withdrawal is created.

Without careful engineering, this can expose revenue leakage if there is even the slightest synchronisation delay or lack of transaction locking. By example:

A tenant has their own SoV and keeps it in sync with Eclipse by using Eclipse webhooks to be notified of Eclipse transactions. For the opposite scenario, when the tenant has transactions they send these to Eclipse API's just after they have happened. A customer has R1000 balance in a wallet on both Eclipse and the tenant and everything is in sync. The customer generates a CashXpress ATM withdrawal token for R1000. They go to an ATM and as they submit the withdrawal, they also at that same moment do a P2P transfer in the tenants App for R1000 to a friend. The withdrawal request hits Eclipse and is debited R1000. The P2P hits the tenants SoV that is not yet notified by webhook and it transfers R1000 to the friend's wallet. The customer walks away with R1000 in cash and their friend has R1000 in their tenant SoV and R0 in the Eclipse wallet. The systems are now out of sync as Eclipse would refuse to transfer the R1000 to the friends wallet on Eclipse as the source has insufficient balance. The friend could possibly spend the R1000 they have on goods available within the tenants App.

For this reason, the recommended approach to keeping wallets synchronised is to add the tenants SoV as a wallet type in Eclipse and have Eclipse delegate all ledger transactions to

the SoV. The tenant can then utilise all of the Eclipse API's and know that wallet movements will automatically happen against their ledger without them needing to listen for webhooks nor process event streams. Eclipse takes care of the complexities of dealing with calling the SoV and ensuring failures and timeouts are handled correctly without the possibility of lost transactions.

In order for a tenant's Store of Value to be plugged into Eclipse, the SoV needs to expose APIs that Eclipse can call in real time during transactions. These API's are as follows:

Mandatory API's for Tenant to expose

- Create a new wallet and return a unique identifier for that wallet that will never change
- Get a wallet's current and available balance. Available balance is the current balance less any reservations
- Get the transaction history (ledger movements) for a wallet between 2 dates
- Transfer funds between 2 wallets in a debit/credit pair. While ensuring that:
    - A debit will fail if the amount exceeds the available balance
    - All the transfers debit/credit legs obey all rules of an ACID transaction
    - A transfer should be passed an idempotency key and should fail with an error if a transaction with this key already exists on the ledger (to avoid duplicates/retransmissions)

In addition to the mandatory API's, wallets need to be able to reserve/lock funds for long running transactions. This can either be done by the tenant and expose reservation API's, or if the tenants wallet does not support reservations, then Eclipse can cater for this by debiting the wallet and storing the funds in a special reservation wallet that is like a suspense account. If the SoV does support reserving funds, then the following API's are required to be exposed:

Reservation API's

- Lock a certain amount in a wallet with a description for the lock, a session identifier for the lock and an expiry date for the lock. When the expiry date is reached, the funds should no longer be locked. When one or more reservations are created, they should sum up to the difference between a wallet's current and available balance. Transfers should not be able to exceed the available balance.
- Get the list of reservations placed on a wallet
- Delete a reservation

- Support doing a transfer where if the session id of a reservation is included as a field in the transfer then the reservations with that session id will be deleted initially as part of the transfers ACID transaction on the SoV

Assuming the tenants store of value supports these API's, the Eclipse team can write a specific wallet implementation to point to the tenants store of value.

A due diligence would however need to be done to ensure the tenants SoV can cater for their envisaged transaction throughput while maintaining a latency of less than 100ms for all transactions (excluding network overhead).

Tenant Config Reference

Config name	Description	Example
institutionName	Used for tenants that need card issuing. This is the name of the sponsor bank and associated config that should be used for card issuing. It determines the various certificates and configs used when calling PTS.	ZA_Nedbank
acceptorTerminalId	PTS configuration. Will be set up by back office support for the tenant	ABCDEFGH
acquirerCountryCode	The ISO country code of the tenant	710
acquiringInstitutionCode	PTS configuration. Will be set up by back office support for the tenant	020353
pointServiceCodes	PTS configuration. Will be set up by back office support for the tenant	000000000070035690210
posEntryMode	PTS configuration. Will be set up by back office support for the tenant	010
posPinCaptureCode	PTS configuration. Will be set up by back office support for the tenant	00
mcc	PTS configuration. Will be set up by back office support for the tenant	5999
halo.jwt.issuer	The JWT issuer to set in tokens generated for the Halo tap on glass SDK	EclipseHaloTest
halo.jwt.subject	The JWT subject to set in tokens generated for the Halo tap on glass SDK	Eclipse
halo.jwt.audience	The backend server that will receive Halo JWT's	https://kernelserver.haloplus.io
halo.jwt.aud_fingerprints	Certificate fingerprints for Halo	1,2,3
masterpass.config.apiUser	The API username to use when generating and	merchant-1

	looking up QRCodes on masterpass	2345
masterpass.config.apiPassword	The API password to use when generating and looking up QRCodes on masterpass	1A2B3C
masterpass.config.aesKey	The AES encryption key to decrypt incoming payment notifications from Masterpass	DAB448751C53A8F91AA191C34001E501
impersonationAllowed	Whether a JWT for a tenant system role can call login for another identity under the tenant without providing a password and be given a JWT representing that identity. Useful when a tenant needs to log in on behalf of a customer without knowing their password.	false
fees.amount.config.Wdr.ZA_PAYCORP_ATM.ATM	The fee to charge a wallet withdrawing from a Paycorp ATM. The fee can be percentage based (P) or absolute (A) or a combination. E.g. 2.5A would mean R2.50 fee while 0.5P would mean half a percent of the amount and 2A,1P would mean R2 plus 1 percent	10A
fees.amount.config.Wdr.ZA_PNP.CASH	The fee to charge a wallet withdrawing from PnP. The fee can be percentage based (P) or absolute (A) or a combination. E.g. 2.5A would mean R2.50 fee while 0.5P would mean half a percent of the amount and 2A,1P would mean R2 plus 1 percent. The fee is charged when the withdrawal is made.	10A
fees.amount.config.Wdr.ZA_PNP.TENDER	The fee to charge a wallet paying for goods at PnP using a TENDER token. The fee can be percentage based (P) or absolute (A) or a combination. E.g. 2.5A would mean R2.50 fee while 0.5P would mean half a percent of the amount and 2A,1P would mean R2 plus 1 percent. The fee is charged when the payment for goods is made.	10A
fees.amount.config.Pay.ZA_PNP_DEPOSIT	The fee to charge the receiving wallet when it is credited by a deposit done at PnP. The fee can be percentage based (P) or absolute (A) or a	10A

	combination. E.g. 2.5A would mean R2.50 fee while 0.5P would mean half a percent of the amount	
fees.amount.config.Pay.GLOBAL_BANK.EFT.SERVER.ZA_Nedbank	The fee to charge a wallet receiving an EFT from Nedbank. The fee can be percentage based (P) or absolute (A) or a combination. E.g. 2.5A would mean R2.50 fee while 0.5P would mean half a percent of the amount. The fee is charged to the recipient wallet as and when the payment is credited to that same wallet.	10A
fees.amount.config.Pay.GLOBAL_BANK.ATM.SERVER.ZA_Nedbank	The fee to charge a wallet receiving a ATM deposit done at a Nedbank ATM. The fee can be percentage based (P) or absolute (A) or a combination. E.g. 2.5A would mean R2.50 fee while 0.5P would mean half a percent of the amount. The fee is charged to the destination wallet as and when the credit is processed on that same wallet.	10A
fees.amount.config.Pay.GLOBAL_PAYMENT_LINK.LINK.CLIENT	The fee to charge a wallet for creating a payment link. The fee can be percentage based (P) or absolute (A) or a combination. E.g. 2.5A would mean R2.50 fee while 0.5P would mean half a percent of the amount	10A
fees.amount.config.Pay.GLOBAL_WALLET.WALLET	The fee to charge for making a payment with an Eclipse wallet. The fee can be percentage based (P) or absolute (A) or a combination. E.g. 2.5A would mean R2.50 fee while 0.5P would mean half a percent of the amount	10A
fees.amount.config.Pay.ZA_MASTERPASS_IN.CARD.SERVER	The fee to charge the receiving wallet when it is credited by a payment coming in through Masterpass (could also be a topup). The fee can be percentage based (P) or absolute (A) or a combination. E.g. 2.5A would mean R2.50 fee while 0.5P would mean half a percent of the amount	2.5P
fees.amount.config.Wdr.ZA_NEDBANK_EFT.EFT	The fee to charge a wallet to EFT funds out via Nedbank CPS. The fee can be percentage based (P) or absolute (A) or a combination. E.g. 2.5A	10A

	would mean R2.50 fee while 0.5P would mean half a percent of the amount	
withdrawals.limits.grouping	What to group by when calculating the monthly and daily withdrawal limits. Can be userId (Default. calculate based on the userId that own the wallet doing the withdrawal) or deliverToPhone (calculate based on the phone number the withdrawal is being done for)	deliverToPhone
withdrawal.limits.monthly	The max amount allowed to withdraw across any cash withdrawal mechanism since the start of the calendar month.	24000
withdrawal.limits.daily	The max amount allowed to withdraw across any cash withdrawal mechanism since the start of the UTC day.	3000
source.wallet.config.ZA_MASTERPASS_IN.CARD.SERVER	Wallet to debit when a Masterpass payment is received	111
source.wallet.config.GLOBAL_BANK.EFT.SERVER.ZA_Nedbank	Wallet to debit when a Nedbank EFT is received	111
source.wallet.config.ZA_PEACH_CARD	Wallet to debit when a Peach topup is done	111
source.wallet.config.ZA_PNP_DEPOSIT	Wallet to debit when a PNP cash deposit is done	111
destination.wallet.config.ZA_PAYCORP_ATM.ATM	Wallet to credit when a Paycorp withdrawal is done	111
destination.wallet.config.ZA_PNP.CASH	Wallet to credit when a PNP withdrawal is done	111
destination.wallet.config.ZA_PNP.TENDER	Wallet to credit when a PNP payment for goods is done	111
destination.wallet.config.VAS.X	Where X is the provider Id of the VAS integration	111
fees.wallet.config.Wdr.ZA_PNP.CASH	Wallet to credit with fees when a PNP withdrawal is done	111
fees.wallet.config.Wdr.ZA_PNP.TENDER	Wallet to credit with fees when a PNP token is used to pay for goods	111
fees.wallet.config.Wdr.ZA_PAYCORP_ATM.ATM	Wallet to credit with fees when a Paycorp withdrawal is done	111

fees.wallet.config.Pay.GLOBAL_BANK.EFT.SERVER.ZA_Nedbank	Wallet to credit with fees when a Nedbank EFT deposit is done	111
HMACOutboundSignatureKey	Secret key to use for generating HMAC signatures on webhooks. If not provided then signatures wont be added on WebHooks	MyBigSecret1
HMACInboundSignatureKey	Secret key to use for generating HMAC signatures on calls into Eclipse. If not provided then Eclipse will not verify message integrity	MyBigSecret2
HMACInboundSignaturePositions	Comma delimited list of tenant admin user positions that will be required to have an HMAC signature header if HMACInboundSignatureKey is set	TENANT_SYSTEM
xTenantOutboundWalletId	Wallet ID of a system wallet through which all transfers from the tenant to another tenant will flow	111
xTenantInboundWalletId	Wallet ID of a system wallet through which all transfers from another tenant into this tenant will flow	111
nedbank.cps.config.clientProfile	Profile identifier for withdrawals when using ZA_NEDBANK_EFT. Provided by Nedbank	1234
nedbank.cps.config.nominatedAccount	Debit account for withdrawals when using ZA_NEDBANK_EFT.	1111111111
nedbank.cps.config.chargesAccount	Charges account for withdrawals when using ZA_NEDBANK_EFT.	2222222222
nedbank.cps.config.environment	Environment (P) Production	P
nedbank.cps.config.clientIdentifier	Client identifier for withdrawals when using ZA_NEDBANK_EFT. Provided by Nedbank	ABC
mobilerecharge.providers	VAS property: We can define the different VAS providers by comma separated.	com.ukheshe.services.mobilerecharge.provider.MasterpassRecharg

		eProvider, com.ukhesheservices.mobilerechargeprovider.TelkomINMobileRecharge
initialOCRChecks	Kyc/kyb property:Define the list of initial OCR checks that must be passed before doing the above compliance checks. If the value of this property is not set or empty then eclipse will do the compliance checks even if the initial OCR checks are failed. We can define the value of this property as comma separated. We can used "|" for or condition(Please see the example value).	identityNumberMatchesNationalIdentity|passportNumberMatchesPassport, selfieMatchesNationalIdentity|selfieMatchesPassport
syncCreditBureauName	Kyc/kyb property: If this property is not empty then eclipse will do credit bureau checks. We can define a list of credit bureau checks by comma separated.	securecitizen
complianceRiskAgencies	Kyc/kyb property: If this property is not empty then eclipse will do the sectionlist compliance checks. We can define the sectionlist agencies by comma separated i.e OFAC,UN,MATCH	OFAC,UN,MATCH
document.type.config	Kyc / Kyb Global property . Doing a brute force config driven matching check for "nationalIdentityIsANationalIdentity" and "passportIsAPassport". For "nationalIdentityIsANationalIdentity" we only depend on the configuration driven matching check(refer Example value for regular expression). While in case of "passportIsAPassport" we decode the passport and check digit are correct and if not then we do the config driven match.	NId=(Identity Number|I.D. No.).(\s*+([0-9]+\s*+)+) Passport=< regular expression >

	Note: This property must be set on a global level not a tenant level.	
allowSelfieCheckWithNationalDatabase	Kyc property. Used to check "selfieMatchesNationalDatabase" . If the value of this property is true then it will compare the selfie provided with the Photo ID that the national database(i.e secured citizen) have on record. This property is dependent on the value of the "syncCreditBureauName" property. if value of syncCreditBureauName is not empty and allowSelfieCheckWithNationalDatabase=true then it will compare else not.	true/false. Default is true.
allowDuplicateIdPassport	Used to allow duplicate passport or id number for tenant. Value=true, will allowed duplicates.	true/false Default value is false.

The following config values do not sit within the tenant configuration but do impact the tenant:

Config name	Description	Example
mustache.sms.withdrawal.<withdrawalType>.<tenantId>

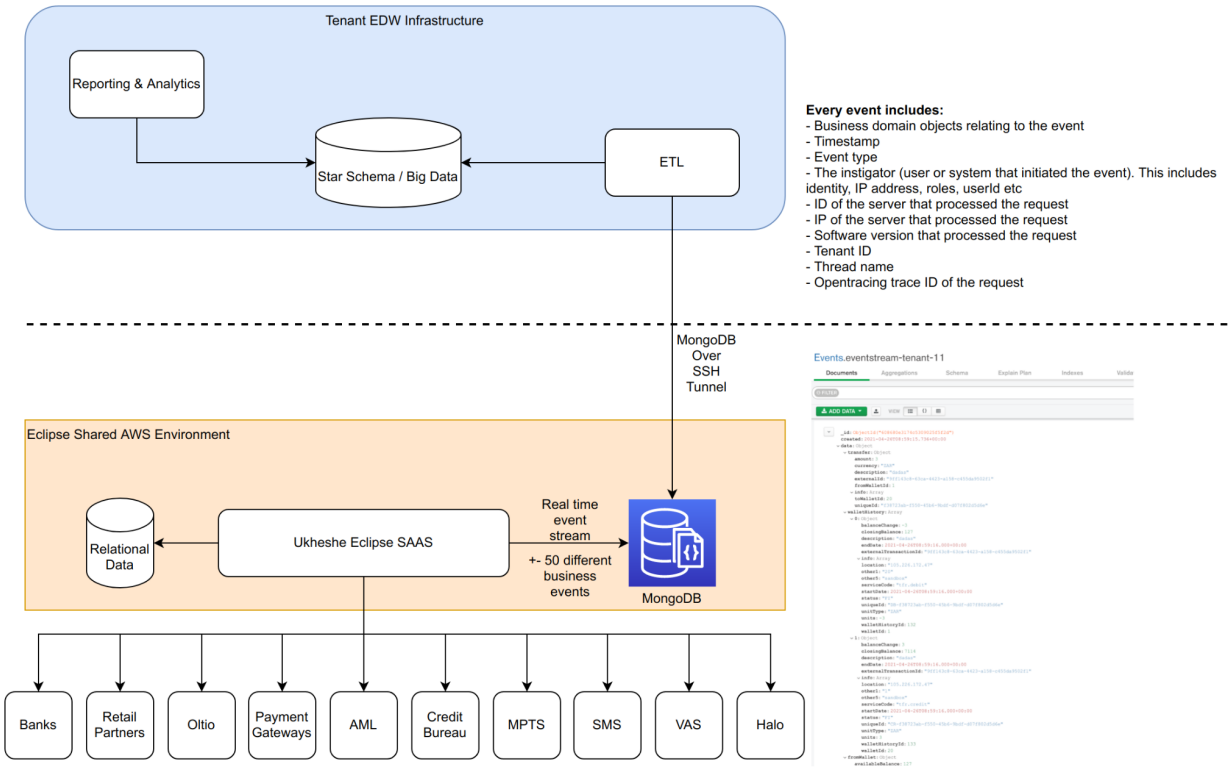
E.g. mustache.sms.withdrawal.ZA_PAYCORP_ATM.10 | Mustache template for sending an SMS to the deliverToPhone number when the withdrawal is created. If not configured, no SMS is sent | Your withdrawal token is 12345 |
| | | |
| | | |

Reporting

Reporting in Eclipse can be approached in one of two ways:

1. All Creates, Updates and Deletes across the system are persisted in JSON in a document database (MongoDB compliant) under a single collection per tenant. A tenant can request a user capable of connecting to the collection and can then stream and process all events into their own EDW or data analytics.
2. Eclipse has a rudimentary ad hoc reporting API for tenants that want to access ad hoc reports without needing to process the JSON data themselves.

Document Database

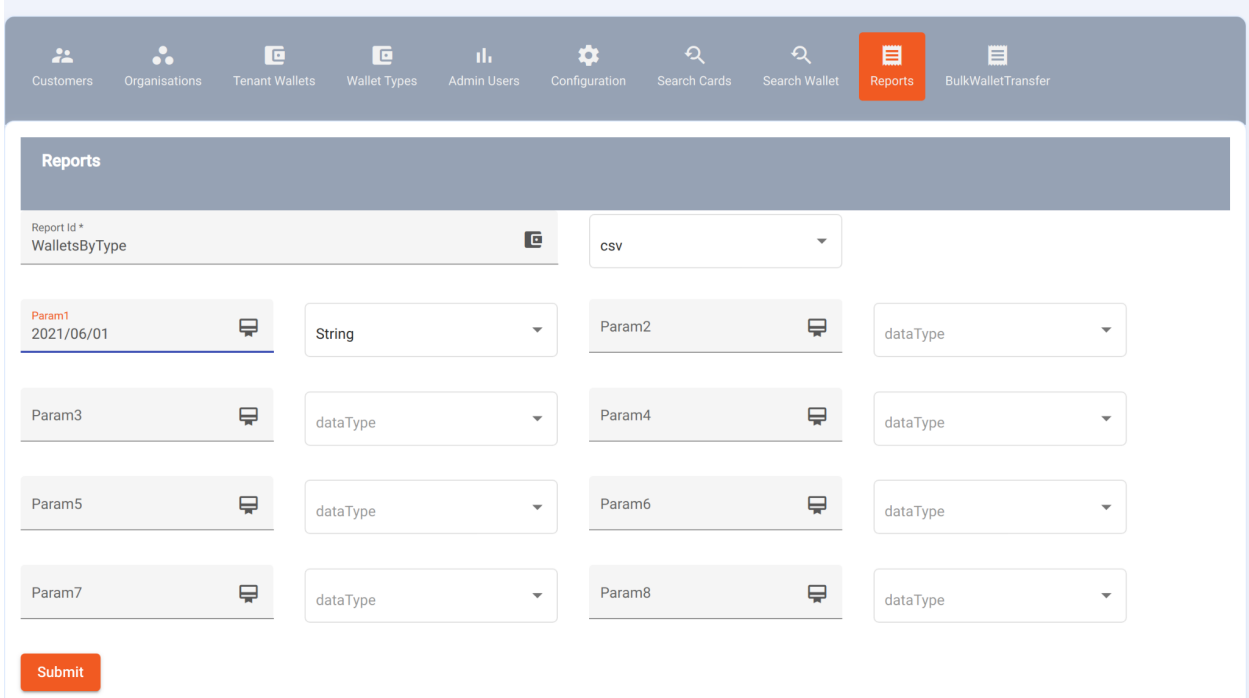


A closer view of a typical event in Mongo:

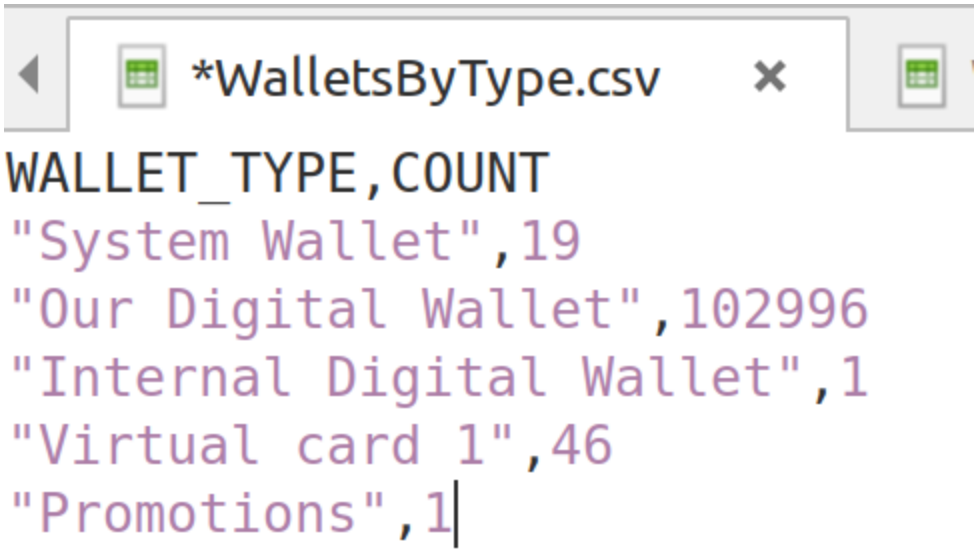
Ad Hoc Reports API

The reports API is available at the tenant level and is used to run pre-configured reports that take pre-agreed parameters and return CSV, JSON etc:

The API can be used via the admin portal as well:



In order to have a report set up, the tenant must provide a detailed definition of what data the report must contain (columns) and if parameters should be passed, what these would be. E.g. a tenant could request a report be added called “WalletsByType” that accepts a single parameter being a date and shows the number of wallets per type created since that date. The report output can be CSV, JSON or XML. Example CSV:



Postman

We have a Postman collection available at
https://www.getpostman.com/collections/c92fd8d5b42fa3d87ad5

Why use Postman

Get up and running in seconds

Instead of creating calls manually to send over the command line, all you need is a Postman Collection - Conveniently located above.

Parameterize your requests

Use our built-in library of dynamic variables to generate dynamic dummy data for testing multiple iterations of the same request with different values.

Inspect responses

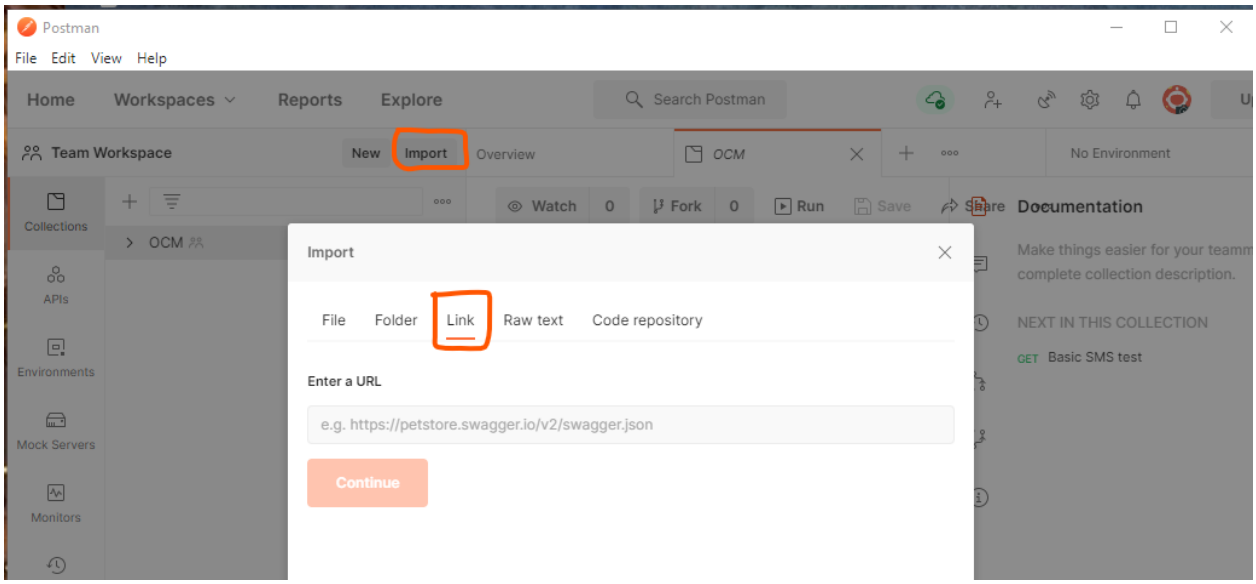
View the status code, response time, and response size. Postman's automatic language detection, link and syntax highlighting, search, and text formatting options make it easy to inspect the response body.

Code Snippet Generation

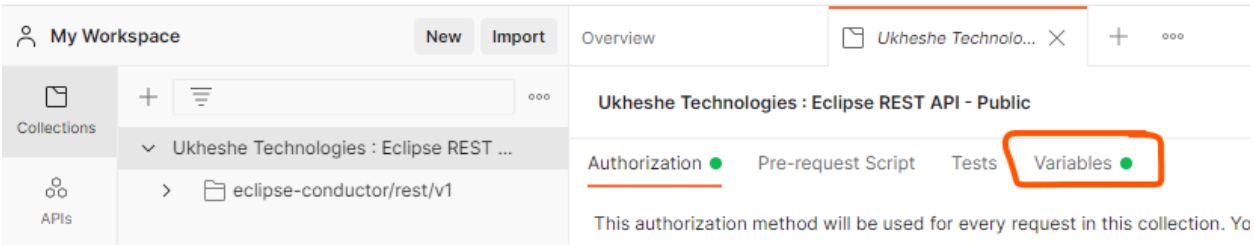
Postman offers a great tool to generate code snippets to help your devs to get going.

How to use

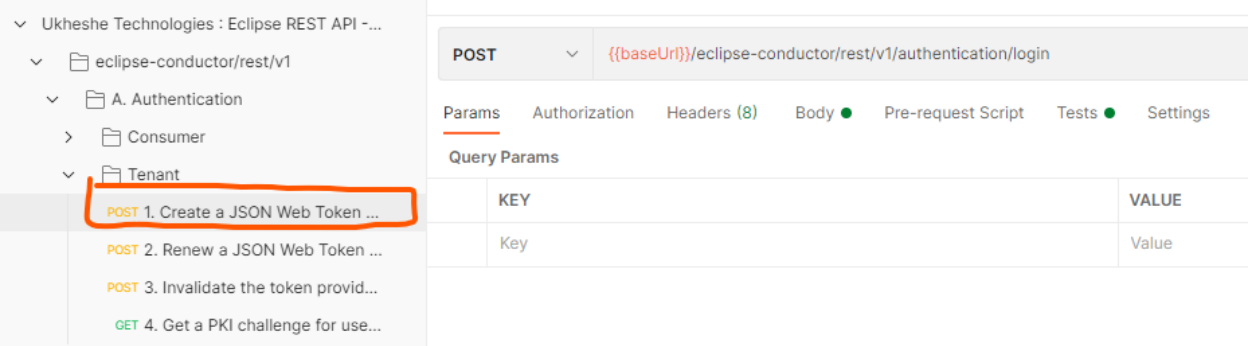
1. Open Postman and Import the Eclipse Postman Collection above.



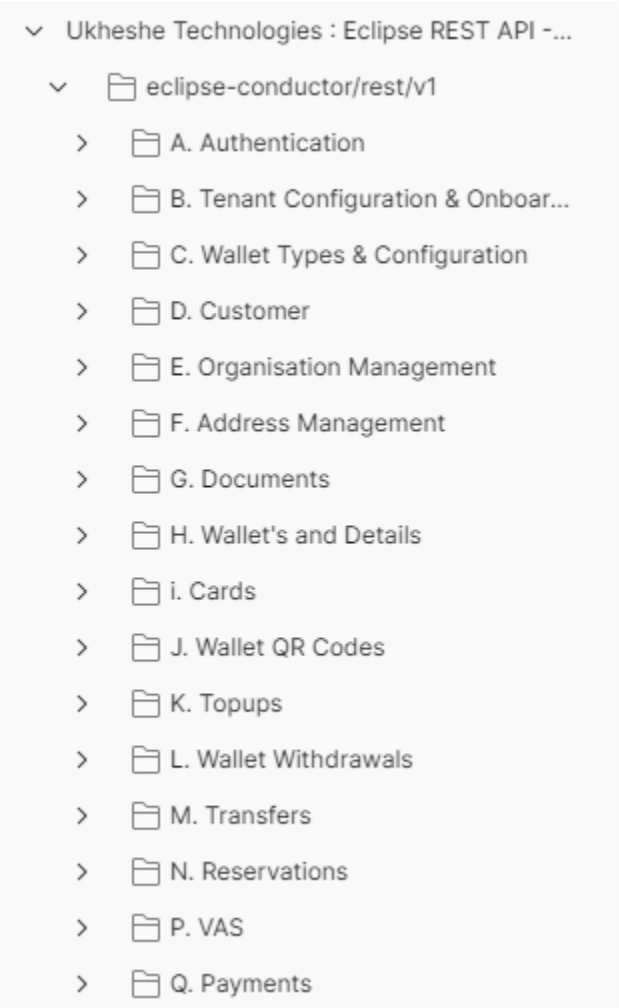
2. Set your collection variables
   a. Click ‘Ukheshe Technologies : Eclipse REST API - Public’ collection
   b. Click variables tab
   c. Enter tenantIdentity variable
   d. Enter tenantPassword variable
   e. *Many more predefined variables are located here, many will be automatically populated after some requests are made, using predefined test cases*



3. Run tenant auth request, this will persist your jwt token to a variable for all next requests. (needs to be redone when jwt expires)



4. Now you can go ahead and try any of the Eclipse API's



Application Error Codes

Any HTTP response with a status code of 400 or greater will contain an array of one or more errors that cause the request to fail.

```
[
  {
    "code": "USR007",
    "description": "Incorrect code",
    "severity": "LOW",
    "type": "BUSINESS",
    “traceId”: “hchuewhuiehdiwhei”
  }
]
```

Error codes are of the format AAANNN where AAA is a code in A-Z describing the error category and NNN is a numeric providing a more specific error within the category.

The following table contains the descriptions of the possible error codes that may be returned

Code	Description
ENF001 ENF002	Entity Not Found. Typically accompanied with an http status code 404. Indicates that the resource being referred to could not be found.
RLT001	The request is rejected due to a Rate Limit being hit. Try again later when the next rate limit window opens
DAE001	A Web Application Exception was reported from a downstream integration below Eclipse. The exact reason is not known but the description may contain more details
UNK001	An Unknown error has occurred. The exact reason is not known but the description may contain more details
NUR001	Something during request processing expected a single result for something but instead got more than one
PNF001	A configuration property is missing. The property is required to process the request. Contact Eclipse support to understand what configuration is needed.

VAL001	A data constraint violation has occurred.
JPE001	An error occurred parsing the JSON provided in the request
SEC001	A security exception has occurred. The request does not have the required rights/access to do what it is trying to do or authentication data is invalid
NFE001	A number has been provided in an incorrect format.
IAE001	A parameter provided in the request is invalid in the context of the request. The description will explain what parameter is invalid. This is alerted to the Eclipse operations team automatically.
IAE002	A parameter provided in the request is invalid in the context of the request. The description will explain what parameter is invalid. These are typically from fields provided by the end user and are not alerted to the Eclipse operations team.
ISE001	The request could not be processed as something is not in a state that allows the changes that are being requested.
OLE001	An optimistic lock exception has occurred. Typically caused by a request to update something which has been updated by another request since the data was last read.
SQL001	An error was reported by a database trying to persist the data of the request
IOE001	An I/O Exception occurred either accessing data over the network or a file system
WAL001	The request could not be processed as a wallet involved in the request did not have sufficient balance to allow the transfer/charge.
USR010	Incorrect challenge response
USR004	Password is too weak
USR006	Password change code is incorrect
PAY002	Invalid data in a payment request caused the payment to fail
PAY005	An unknown error prevented the payment from being processed within Eclipse
UOE001	An unsupported operation has been attempted. E.g. trying to do

	something on a gateway that does not support that functionality.

FAQ

There is no mention of merchants in the document

On Eclipse there is no concept of a merchant but rather a customer or organisation that can have access to different functionality .

For the KYC process what are the mandatory fields/documents required? Are there minimum required documents to complete this process or are all the specified documents required?

Running a ratify will result in Eclipse looking at the profile, address and attached documents and running all possible checks based on what is present and then returning an analysis of the findings:
https://eclipse-java-sandbox.ukheshe.rocks/eclipse-conductor/docs/#/d)%20Customer%20Management/post_eclipse_conductor_rest_v1_tenants__tenantId__customers__customerId__ratify

KYC document limits

Max size for all documents under a profile is 20MB in total.

Max size of an individual document (FACIAL_PHOTO) that can be ratified is 5MB.

For the KYC ratify, what combination of true fields are required to pass KYC successfully?

What really matters is what the walletType requirements are for any customer to be given a wallet of a certain type. On Eclipse it needs to be fairly dynamic as different countries have different rules etc so we can't have any fixed enums/statuses for this. In

SA, assuming R3k daily and R25k monthly limit on a wallet, we would set the config on the walletType you will use to require the following to have passed as true on ratify:

firstNameMatchesNationalIdentity

identityNumberMatchesNationalIdentity

lastNameMatchesNationalIdentity

selfieIsASelfie

selfieMatchesNationalIdentity

If those are not true then we would return an error when trying to give the customer a wallet.

And in order to get those to all pass, you need to have set nationalIdentityNumber, firstName, lastName on the customer's profile, and attached a selfie and picture of national ID as the 2 documents.

You then interact with the customer on your App/Frontend to get data until those tests pass and when they do then you know you can go ahead and add the wallet.

NOTE: "selfieIsASelfie" is a picture of a person with thumbs up.

How can an organization get funds into an Eclipse wallet?

They simply EFT to the agreed Nedbank bank account that acts as their suspense account, using the 8 digit wallet reference of the wallet they want to be credited with the amount they EFT.

The exact Nedbank account would be agreed as part of the onboarding with finance.

What are the Eclipse production IP's that do callbacks/webhooks in case I want to whitelist them?

54.220.185.240, 79.125.40.127, 18.134.2.182, 3.11.195.109

Note: Sandbox IP's are dynamic.

What is the convention of the "amount" field in various API's

Amount in the major currency unit e.g. 50.80 for 50 GBP and 80 pence. E.g. 10.88 in ZAR is 10 Rand 88 cents

When providing a callbackUrl, what happens if the call from Eclipse to the callback URL fails? Will it be retried?

Eclipse callbacks are guaranteed to retry forever until an HTTP status code of < 300 is returned by the tenants URL. Eclipse retries on an exponential backoff policy that starts at once per second for 2 tries, then once every 10 seconds for the next 3 tries, then once every 2 minutes for the next 5 tries then once every 2 hours for 12 tries then once a day for a week and then once a week for a month and finally once a month from then on.

What the process of Topups at retailers (`ZA_PNP_CASH`)Topups

After a 'completionToken' is generated on the Topup API, the following steps need to be followed:

Customer Message:

> *Please give Rxx.xx to a Pick n Pay Teller along with reference number '{completionToken}'. The amount will immediately reflect on your wallet.*

Teller Instructions:

1. *From the menu select 'Value Added Services*
2. *Select 'Mobile"*
3. *Select 'Deposits'*
4. *Select 'uKheshe'*
5. *Insert reference number*

What the process of Withdrawals at retailers (`ZA_PNP_CASH`)

After a 'token' is generated on the Withdrawal API, the following steps need to be followed:

Customer Message:

Present token, amount and expiry to customer

Teller Instructions:

1. *From the menu select 'Value Added Services*
2. *Select 'Mobile'*
3. *Select 'Withdrawals'*
4. *Select 'uKheshe'*
5. *Insert reference number*

Sandbox Testing Accounts

Wallet Withdrawals

ZA_NEDBANK_EFT

Account Numbers:

1001004345 | 1201303338 | 1201304067

Branch code: 198765

Withdrawal Response simulation. (Sandbox only)

You can simulate success and error responses for all withdrawal types, kindly follow below steps.

Step1: Initiate withdrawal:

https://eclipse-java-sandbox.jini.rocks/eclipse-conductor/docs/#/l)%20Wallet%20Withdrawals/post_eclipse_conductor_rest_v1_tenants__tenantId__wallets__walletId__withdrawals

Step2: simply call the API below, with status `SUCCESSFUL, ERROR`

*https://eclipse-java-sandbox.jini.rocks/eclipse-conductor/docs/#/l)%20Wallet%20Withdrawals/put_eclipse_conductor_rest_v1_tenants__tenantId__wallets__walletId__withdrawals__withdrawalId_*

Wallet Topups

ZA_MASTERPASS_CARD

Updated 6 January 2021

5520578440225402 22/11 804 test123

4172900030027739 25/01 383 test123

5520578440225402 22/11 804 test123

ZA_PEACH_CARD

Card Number 4242424242424242 with any future expiry date and CVV 123

Masterpass QR test cases

Sandbox

Please make sure that you support as many of these use cases as possible.

https://docs.google.com/document/d/1CA_KiNx89isNrIuVPXu8HocXQ_-Ydu4m/edit?usp=sharing&ouid=105463204992414282726&rtpof=true&sd=true

Production

https://drive.google.com/file/d/15IbKXPQhqruIopQhsAlMPMrYzUDsZuNV/view?usp=sharing

For Reference on handling QR code use cases, please refer to any Masterpass app in the market, see doc below how to switch the app to sandbox environment.

https://drive.google.com/file/d/1Oo9-H4aG1_Bz-SITIy9hMSbQQnUz2xoW/view?usp=sharing

Test Cards for Acquiring

Visa VTS (tokenisable cards)

4895 3799 8701 2927 (exp: 12/22, CVV: 525)

4895 3799 8701 2935 (exp: 12/22, CVV: 862)

Mastercard M4M (tokenisable cards)

2223 0015 0000 0018 (exp: 11/21, CVV: 123)

2223 0015 0000 0026 (exp: 11/21, CVV: 123)

Supported Fees Setups

How to setup fees for Wallet to wallet or wallet to physical transfer within tenant?

Eclipse supports these fees on the wallet type. We need to add "*sendTransferLogicSetId*" configuration on the wallet type. Value for this is a Predefined Id (numeric) defining a set of

logic to run whenever a debit or credit happens on a wallet. Pre-defined Id obtains from "wallet_logic_set" table in eclipse. We can ask the development team to add an entry into this table for custom (fees)logic is needed by the tenant.

Note: It only triggered when transfer gets done and it applicable to all wallet which has been created under this wallet type.

How to restrict customer to do a transfer to another customer Digital wallet if Digital wallet does not have minimum balance?

We can add this restriction as configuration on wallet type. We need to add configuration "*minimumBalance*" and numeric value. The balance below defined numeric value, wallet cannot be debited for normal payments and transfers. Only applicable to digital wallets.

How to setup monthly fees on Digital wallets/ Physical/virtual cards?

Monthly fees setup can be done on wallet type configuration. We need to add "*monthlyFeeLogicSetId*". Value for this is a Predefined Id (numeric) defining a set of logic to run once per wallet on the first of each month. Pre-defined Id obtains from "wallet_logic_set" table in eclipse. We can ask the development team to add an entry into this table for custom logic is needed by the tenant.

How to setup withdrawlas fees?

We can setup the withdrawls fees under the tenant config (tenant.config.<tenantId>). *Format:"fees.amount.config.Wdr."<GateWay>.<Withdrawal Type>* i.e. "*fees.amount.config.Wdr.ZA_PAYCORP_ATM.ATM*" The fee can be percentage based (P) or absolute (A) or a combination. E.g. 2.5A would mean R2.50 fee while 0.5P would mean half a percent of the amount and 2A,1P would mean R2 plus 1 percent. The fee is charged when the withdrawal is made. For more example please refere the ***Tenant Config Reference section.***

How to setup fees on VAS purchase through digital wallet.

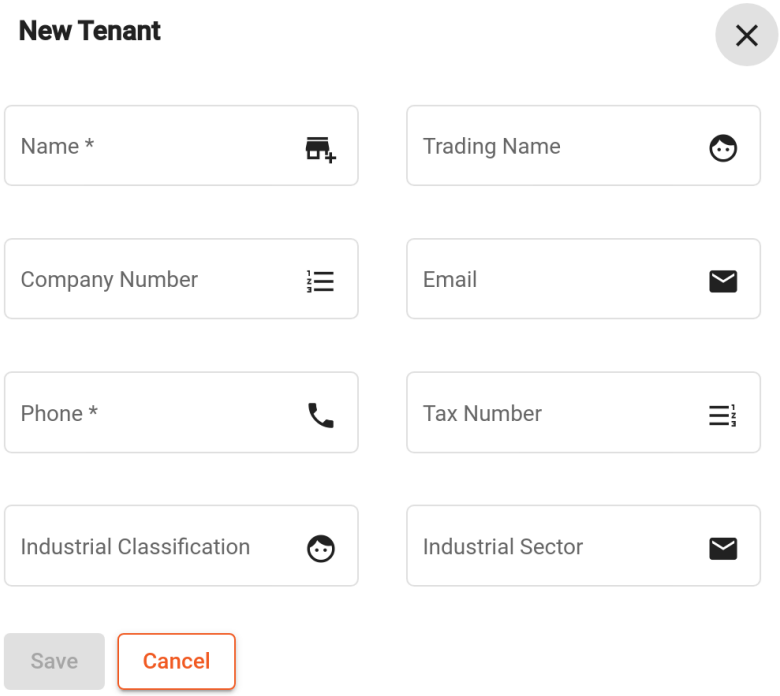
Eclipse does not charge the specific fees on VAS purchase. If tenant want they can configured on their side.

Appendix 1: Production Setup Prerequisites

In order for a new tenant to be set up on the production system, various prerequisites must be met to ensure a smooth launch. These prerequisites depend on the Eclipse functionality that will be used.

Functionality Dependent Prerequisites

Text highlighted in yellow indicated prerequisites to be provided prior to commencement of the tenant setup in production. Other prerequisites are things that should be set up prior to the tenant going live

Functionality	Prerequisites
Any	The following company info is required for creating the tenant: 
Any when launching	• A Nedbank corporate bank account

in RSA	• A Nedbank pool account created by Ukheshe specifically for the Tenant • Details on doing the above can be obtained from Finance operations
Generation of ZA_MASTERPASS QR Codes	• The tenant must be signed up as a Masterpass merchant • The masterpass merchant setup must settle to the Nedbank acquiring pool account • API access must be enabled in the masterpass portal Then when the tenant is being configured in production: • The notification key must be set in the tenants configuration as masterpass.config.aesKey • API access enabled in the masterpass portal and the user and password set in config properties masterpass.config.apiUser and masterpass.config.apiPassword • The masterpass notification URL must be set to https://ukheshe.live/eclipse-conductor/rest/v1/masterpass/incoming-payments?configId=XXX where XXX is the tenantId in production
Any	• The RBAC spreadsheet must be completed with the tenant and signed off and understood. This is simple enough for tenants only accessing the API's with TENANT_SYSTEM role but is fairly detailed when allowing end users to call Eclipse APIs with their own identities
Any	• Tenants must use PKI authentication for TENANT_SYSTEM access. If they choose not to then an email must be sent notifying them of the risk of their password being leaked and confirmation received that they accept this risk.
Mastercard Card issuing (virtual or physical)	• The process/requirements to launch card programs are beyond the scope of the integration guide and are dealt with by the issuing team in detail. The outcome of that process is the Mastercard program Code e.g. PRG55 that must be configured as ProgramCode on a Card wallet type set up on the tenant. Then when the tenant is being configured in production: • Config params must be set up: institutionName, acceptorTerminalId, acquirerCountryCode, acquiringInstitutionCode, pointServiceCodes, posEntryMode, posPinCaptureCode, mcc • A virtual card should be set up to hold any reservations on

	other cards. The wallet id of this card should be set as config element reservationWalletId
Any when launching in RSA	• As Ukheshe is the accountable institution, Eclipse must hold all KYC data and run KYC checks even if pre-done by the tenant. • Before moving to production, the sandbox wallet types must be configured to run KYC checks and the tenant must demonstrate that all is working.
Withdrawals of type ZA_NEDBANK_EFT	• Nedbank must set up a CPS profile and provide the information required to populate the config values nedbank.cps.config.*
Any	• If any fees are to be charged by Eclipse for withdrawals and payments then these should be specified. Then when the tenant is being configured in production: • The config elements fees.amount.config.* should be set up. If not provided they default to 0. If provided then config elements fees.wallet.config.* should be set up to specify where the fees should accumulate.
ZA_PNP_CASH withdrawals, ZA_PNP_TENDER withdrawals, ZA_PNP_CASH topups	• A system wallet must be set up under the tenant for PNP topups and withdrawals. Ukheshe will do this.
VAS Catalog & Payments	• Accounts must be created or configured on the applicable VAS partners and system wallets set up to be credited when VAS is purchased
Message Integrity	• HMACOutboundSignatureKey HMACInboundSignatureKey HMACInboundSignaturePositions should all be set. • A good way of generating a random secret for HMAC is: openssl rand -base64 33

Technical Prerequisites

- The tenants sandbox access logs will be checked to ensure the following practices are being adhered to:
    - JWT's are being reused and not obtained too often

- Unnecessary calls, polling etc are not being done
- Callback URLs are being processed
- If card details are being provided to an end user then the tenant must provide an overview of the process they are following and written confirmation that at no point are they calling APIs to get unmasked card details from their servers.