

Oracle® Database

Oracle Blockchain Platform Enterprise Edition Digital Assets



Release 26.3.1

G57244-01

July 2026

ORACLE®

Copyright © 2026, Oracle and/or its affiliates.

Primary Author: Oracle Blockchain Platform development, product management, and quality assurance teams

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle®, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc>.

Oracle customer access to and use of Oracle support services will be pursuant to the terms and conditions specified in their Oracle order for the applicable services.

Contents

1 Overview

Digital Assets	1
----------------	---

2 Using the Blockchain App Builder Extension for Visual Studio Code

Install and Configure the Blockchain App Builder Extension for Visual Studio Code	2
Create and Compose a Digital Assets Project	2
Compile a Contract	5
Deploy a Contract	7
Run a Contract	10
Debug a Contract	13
Upload a Contract Manifest to the Registry	15
Generate a Wrapper API Specification and Postman Collection	16
Generate an RPC Proxy Postman Collection	18
Upgrade the Blockchain App Builder Extension for Visual Studio Code	19

3 Using the Digital Assets Hardhat Command-Line Interface

Install the Digital Assets Hardhat Command-Line Interface	2
Compile a Contract with the CLI	2
Deploy a Contract with the Command Line	3
Upgrade a Contract	6
Upload a Contract Manifest to the Registry with the Command Line	8
Generate a Wrapper API Specification and Postman Collection with the Command Line	10
Generate an RPC Proxy Postman Collection with the Command Line	13

4 Digital Assets APIs

ERC-20 Account Contract APIs	4
ERC-20 Token Contract APIs	13
ERC-1155 Account Contract APIs	25
ERC-1155 Token Contract APIs	31

5 Wholesale Central Bank Digital Currency Application

Install the Wholesale CBDC Application	2
Configure the Rich History Database Profile for the Wholesale CBDC Application	2
Create the Instance for the Wholesale CBDC Application	3
Run the ORDS Script for the Wholesale CBDC Application	4
Wholesale CBDC Deployment Reference	7
One-Click Deploy the Wholesale CBDC Application	11
Configure the Wholesale CBDC Application	12
Wholesale CBDC Sample Application Workflow	14
Central Bank Administrator	17
Central Bank Auditor	19
Central Bank Creator	21
Central Bank Issuer	22
Central Bank Manager	24
Central Bank Retirer	25
Financial Institution Administrator	26
Financial Institution Auditor	27
Financial Institution Manager	29
Financial Institution Officer	29
Financial Institution User	30
Troubleshoot the Wholesale CBDC Application	31

6 Deposit Token Application

Install the Deposit Token Application	2
Configure the Rich History Database Profile for the Deposit Token Application	2
Create the Instance for the Deposit Token Application	3
Run the ORDS Script for the Deposit Token Application	4
Deposit Token Deployment Reference	7
One-Click Deploy the Deposit Token Application	12
Configure the Deposit Token Application	13
Deposit Token Sample Application Workflow	15
Bank Administrator	18
Bank Approver	22
Bank Auditor	22
Bank Creator	23
Bank Issuer	25
Bank Manager	27
Bank Retirer	28
Institution Administrator	29
Institution Auditor	30

Institution Manager	32
Institution Officer	32
Institution User	33
Troubleshoot the Deposit Token Application	34

Index

1

Overview

Oracle Blockchain Platform Enterprise Edition includes features and samples for working with digital assets.

Oracle Blockchain Platform Enterprise Edition also includes an enhanced version of Blockchain App Builder, a tool that assists with rapid development, testing, debugging, and deployment of secure contracts that handle multiple digital assets.

You can use the Blockchain App Builder extension for Visual Studio Code, or the Hardhat command-line interface to work with smart contracts for Besu. Both of these tools support the deployment of mutable contracts for production environments. Mutable contracts can be upgraded if any code changes are needed. If you deploy a smart contract using the RPC proxy API, it is immutable and cannot be upgraded.

Digital Assets

Oracle Blockchain Platform Enterprise Edition includes functions that let you work with digitally native assets.

Digital assets enable organizations to create, manage, and transact tokenized representations of physical and intangible assets. Financial institutions and other businesses can build end-to-end applications quickly and cost-effectively using a scalable and secure architecture. The digital assets functions and samples work on Ethereum instances based on Besu.

2

Using the Blockchain App Builder Extension for Visual Studio Code

The Blockchain App Builder extension for Visual Studio Code helps you build and generate a fully-functional Solidity-based project from a minimal configuration.

The generated project is based on Hardhat 2. After you build a project you can run and test it on a local Besu network or your provisioned Oracle Blockchain Platform network. You can then debug the smart contract or write and run unit tests using the generated code.

Table 2-1 Workflow When Using the Visual Studio Code Extension

Task	Description	More Information
Install and configure	Download the Blockchain App Builder Visual Studio Code extension from your Oracle Blockchain Platform console and install it and any prerequisite software.	Install and Configure the Blockchain App Builder Extension for Visual Studio Code
Create and compose the Oracle Blockchain Platform digital assets project	Select the token type, token standard, token behaviors, and composition mode to generate a Solidity project.	Create and Compose a Digital Assets Project
Compile the smart contract	Compile the contract to generate the bytecode and ABI, which are required to deploy and run the contract.	Compile a Contract
Deploy the smart contract	Deploy the digital assets project locally to the included preconfigured Besu network, local evm node, or remotely to Oracle Blockchain Platform Enterprise Edition for Besu.	Deploy a Contract
Test the smart contract	Test any of the generated methods after your smart contract is running on a network.	Run a Contract
Debug the smart contract	Debug the smart contract by using a simulated development network in Hardhat.	Debug a Contract
Upload the contract manifest to the registry	Upload the generated Besu contract manifests to the Oracle Blockchain Platform registry service to capture the ABI, bytecode, and storage layouts.	Upload a Contract Manifest to the Registry
Generate a wrapper API specification and Postman collection	Generate the wrapper API specification and a Postman collection that contains API endpoints for each contract method.	Generate a Wrapper API Specification and Postman Collection
Generate an RPC proxy API Postman collection	Generate a Postman collection for the RPC proxy calls.	Generate an RPC Proxy Postman Collection

Install and Configure the Blockchain App Builder Extension for Visual Studio Code

You can download the Blockchain App Builder extension for Visual Studio Code from the Oracle Blockchain Platform console.

The following platforms are supported:

- macOS
- Oracle Linux 8.0 or 9.0
- Microsoft Windows 10 or 11

Prerequisites

Before you install Blockchain App Builder on your local system, you must install the prerequisites.

- Node.js version 22.17.0 or later
- npm version 10.9.2 or later

Use nvm to install Node.js and npm. For more information, see [Node Version Manager](#) at GitHub.

Install the Blockchain App Builder Extension

1. Download the extension from the **Developer Tools** tab on the **Blockchain App Builder** pane of the Oracle Blockchain Platform console. On the **Blockchain App Builder** pane, under the **Download** section, select **Visual Studio Code Extension**.
2. In Visual Studio Code, open the **Extensions** panel and then from the **More Actions** menu, select **Install from VSIX**.
3. Locate the downloaded `blockchain-app-builder-for-besu-26.3.1.vsix` file and then click **Install**.
4. Restart Visual Studio Code to complete installation of the extension.

After installation, you can use the Oracle Blockchain App Builder icon on the left side of Visual Studio Code to open the Blockchain App Builder panel.

Create and Compose a Digital Assets Project

You can use Blockchain App Builder to create and compose a smart contract project to work with digital assets.

Blockchain App Builder For Besu generates a self-contained Hardhat project for ERC-20 or ERC-1155 tokens. You choose the token standard and supported capabilities in the extension. The composer function in Blockchain App Builder copies the matching SDK, creates the concrete Universal Upgradeable Proxy Standard (UUPS) token contract, and adds scripts to deploy and upgrade the smart contract.

Use the composer function to create a project where the token ABI and account contract are derived from an explicit capability profile. It does not deploy a token when it creates the project, and it does not run the `npm install` command automatically.

Before you can create and deploy a digital assets project, you must install the Blockchain App Builder for Besu extension and open a Visual Studio Code workspace. To deploy the generated project, Node.js and npm must be installed. You also need an accessible Hardhat or Besu network, an account capable of deploying contracts, and an existing governance contract and governance UUID.

The generated deployment scripts require `GOVERNANCE_ADDRESS` and `GOVERNANCE_UUID` environment variables. The composer function links this governance context to the generated account and token proxies; deployment activation remains subject to the configured governance workflow.

Create a Project

1. Open the **Modules** view in the Blockchain App Builder For Besu activity bar.
2. Select **Open Module Composer** and then select the **Module Composer** tab.
3. In **Create New Hardhat Project**, enter the project name and choose its parent folder. The `compose` function creates the project using the following directory structure:
`<parent_folder>/<project_name>`.
 The project name can contain letters, numbers, periods, hyphens, and underscores, but not double periods (`..`). The project name must start with a letter or underscore and can contain only letters, numbers, and underscores. The destination must be empty. The `compose` function will not overwrite a non-empty project folder.
4. Select the token standard, composition mode, and capabilities, which are described in the following section.
5. Enter a Solidity contract name. For example, enter `DepositToken`.
6. Review the generated composition preview, and then select **Generate Package**.

When project generation is successful, the extension imports the project into the **Contracts** view.

Composition Modes

By default, the `compose` function uses strict composition mode. Both modes are supported for both token standards.

- **Strict composition mode:** The generated token inherits a base that exposes only the selected optional capabilities. To add any capabilities that are not selected during composition, an implementation upgrade is required.
- The generated token inherits a full-surface compatibility base. Runtime configuration controls applicable behaviors, but the broader method surface remains in the ABI.

Compose an ERC-20 Project

Select one required identity-permission module:

- **ERC-5892:** Standard access control back end.
- **ERC-6617:** Bit-based access control back end.

Then, configure the capabilities that are described in the following table.

Input	Values	Notes
Minting	Disabled, Direct, Approval required	Approval required adds the request/approval workflow.
Burning	Disabled, Direct, Approval required	Approval required adds the request/approval workflow.

Input	Values	Notes
Holdable	On or off	Supports hold workflows.
Roles	On or off	Automatically enabled for approval minting, approval burning, or holdable workflows because they require NOTARY authorization.
Daily Limits	On or off	Selects an account variant with or without daily-limit enforcement.
Multi-Level Approvals	On or off	Enables account and approval policies and approval-sequenced hold execution. It automatically enables the holdable capability.

The following behaviors are always part of an ERC-20 generated profile: transferable behavior, delegated operations, pausable behavior, and governance approval flow. Roles are also part of the profile when the compose function has automatically enabled them.

For example, a direct-mint, direct-burn token with no holds generates the strict base `ERC20DirectMintBurnUpgradeable`. Enabling multi-level approvals instead selects a holdable, multi-level approval base and deploys the additional `ERC20MultiLevelApprovalHelperLib` library.

Compose an ERC-1155 Project

ERC-1155 projects use the ERC-5982 access control back end by default. Select one asset type:

- Fungible token: Token classes with a fungible supply.
- Whole NFT: Non-fractional NFT classes.
- Fractional NFT: NFT classes with fractional ownership.
- Combined fungible and non-fungible tokens: Classes for both fungible tokens and NFTs.

Configure direct minting and burning as enabled or disabled. Minting and burning with approvals are not supported for ERC-1155. The compose function always includes transferable behavior, delegated operator approval (`setApprovalForAll` and `isApprovedForAll`), pausable behavior, governance, and ERC-5982 access control.

You can enable roles. For NFT asset types, you can also enable lockable NFTs. Locking is not available for fungible tokens and is disabled when both minting and burning are disabled. Divisible/indivisible behavior is derived from the selected asset type, not entered separately. Daily limits, holdable behavior, and multi-level approvals are not supported.

The combined ERC-1155 profile uses the full-surface `ERC1155CombinedTokenUpgradeable` base even when you select strict composition. Other strict profiles use an asset-type and mint/burn-specific base.

Generated Project Contents

The compose function creates the following project structure.

```
<project>/
├── contracts/
│   └── obp-sdk/
source                                     # copied ERC-20 or ERC-1155 SDK
```

```

|   |   |   <ContractName>/
|   |   |   |   <ContractName>Upgradeable.sol
|   |   |   |   generated.manifest.json          # selected profile and generated
paths
|   |   |   scripts/
|   |   |   |   deploy-<contract-name>/deploy-<contract-name>.ts
|   |   |   |   upgrade/
|   |   |   |   |   upgrade-<contract-name>.ts
|   |   |   |   |   upgrade-<contract-name>-account.ts
|   |   |   .obp-bap/
|   |   |   |   plugins/                                # bundled Hardhat plugin archives
|   |   |   |   project.json                            # Composer project metadata
|   |   |   config/hardhat-env.ts
|   |   |   hardhat.config.ts
|   |   |   package.json
|   |   |   README.md
```

The SDK that is located under `contracts/obp-sdk/` is copied into the generated project and is owned by the project after generation. The manifest records the token standard, composition mode, selected base and account contracts, and the full normalized capability profile

Troubleshooting Generation Errors

Message or Condition	Resolution
Select one identity-permission module	Choose ERC-5982 or ERC-6617 for ERC-20.
Contract name must start with a letter/underscore	Use a valid Solidity identifier.
Project name is required or unsafe name	Supply a simple project name without path separators or double periods (..).
Choose a parent folder	Select an absolute parent folder in the composer.
Target folder exists and is not empty	Choose a new project name or an empty destination. The compose function will not overwrite projects.
ERC-1155 approval-based minting or burning is not supported	Choose Disabled or Direct for ERC-1155 minting and burning.
GOVERNANCE_ADDRESS and GOVERNANCE_UUID are required	Configure both values before running the generated deploy script.

After you generate a contract project, you can use the imported project in the **Contracts** view to compile, deploy, and open deploy or execute forms. Keep the generated manifest with the contract; extension tools use the manifest to pair the generated token and account ABIs with the selected composer profile.

Compile a Contract

After you create and compose a digital assets project, you install dependencies and compile the smart contract.

Install dependencies after you generate or import a project, and install them again whenever the project's package .json file changes. Compile the contract after dependencies are installed and whenever Solidity, TypeScript deployment scripts, or generated project sources change.

1. Open the **Contracts** view in the Blockchain App Builder For Besu activity bar in Visual Studio Code.
2. Select the generated or imported Hardhat project, and then open the **Deploy/Execute** form for that project. If more than one project is available, select the project from the **Project** list at the top of the form.
3. Select the **Deploy** tab, and then select **Install Dependencies**. Wait while the dependencies are installed. If the installation fails, correct the reported error and then select **Install Dependencies** again.
For projects generated with the `compose` function, npm installs the standard Hardhat and OpenZeppelin dependencies and resolves the bundled Oracle Blockchain Platform plug-ins from local archives in the `.obp-bap/plugins/` folder. Keep that folder in the project; the `package.json` file references the plug-in archives by using `local file:dependencies`.
4. After the dependencies are installed successfully, select **Compile**. Wait while the contract is compiled. If compilation fails, correct the reported error and then select **Compile** again.

Hardhat writes compiled artifacts to the project's `artifacts/` directory and uses the `cache/` directory for compiler cache data. The extension uses the artifacts, together with Composer-generated manifests, to discover token and account ABIs for deploy, execute, wrapper API, and RPC proxy tooling.

After you compile the contract, you can configure an environment and then deploy and run the contract.

Troubleshooting Compilation Errors

Message or Condition	Resolution
Unable to run npm install	Node.js or npm is unavailable to Visual Studio Code. Verify the Node.js and npm installation, and then restart Visual Studio Code. If Node.js is managed by nvm, open Visual Studio Code from a terminal so that it inherits the shell PATH variable.
Unable to run Hardhat	Dependencies might not be installed, or local Hardhat, npx, or npm cannot be found. Run Install Dependencies and verify Node.js is available to Visual Studio Code.
npm cannot find a local plug-in archive	Confirm that the <code>.obp-bap/plugins/</code> folder still exists and contains the bundled <code>.tgz</code> files that are referenced by the <code>package.json</code> file. Regenerate the project if the plug-ins folder was removed.
HH1: You are not inside a Hardhat project	Ensure that the correct imported or composed project is selected in the Project selector. The selected root must contain the <code>hardhat.config.ts</code> file.
Contract artifact is missing	Run Compile from the project that owns the contract. Do not use an artifact from a different reference project or SDK standard.
Compile errors remain after a source change	Run Clean , and then run Compile , so that Hardhat rebuilds artifacts from the current sources.

Deploy a Contract

After you compile a smart contract you can deploy it to an Oracle Blockchain Platform Besu environment.

The deployment process creates the required helper libraries and UUPS proxies, links the token and account contexts, and sets the governance context. The token and account are deployed in an inactive state. They become active only through the deployment intent process.

To deploy a compiled contract, obtain the address of the predeployed governance contract and its policy UUID from the digital assets token administrator. Obtain an account with permission to deploy and, where applicable, to submit the deployment intent. The deployment form requires a contract, environment, deployment ID, and registry metadata: smart contract ID, version, description, and author. The deployment ID identifies a deployment state that can be resumed; use a stable, unique value when retrying the same deployment.

Deployment Life Cycle

1. Compile the project.
2. Deploy the Oracle Blockchain Platform digital assets stack.
3. Set account/token and governance contexts. (Proxies are inactive.)
4. Review and submit the deployment intent for both proxy components.
 - No-Op governance: The proposal succeeds and then the components are activated.
 - Governed: The required policy approvals complete and then the components are activated.

Note

Proxies are inactive immediately after deployment. Successful contract deployment does not mean that the token is active. Activation is controlled by the governance contract configured for the deployment, not by a UUPS upgrade.

Create an Oracle Blockchain Platform Besu Environment

An Oracle Blockchain Platform Besu environment stores the Oracle Blockchain Platform RPC host, not a full JSON-RPC endpoint. The extension derives the required Besu and governance routes from this host.

1. Open the **Environments** view, and then select **Add Environment**.
2. Enter a descriptive environment name. For example, enter `obp-test`.
3. Select **Oracle Blockchain Platform Besu** for **Environment Type**.
4. Enter the **RPC URL Host**. For example, enter `https://rpcproxy.example.com:443`. The RPC URL host must be an `http` or `https` origin only, not an endpoint path, query string, fragment, username, or password. For example, enter `https://rpcproxy.example.com:443`, not `https://rpcproxy.example.com:443/v1/besu/proxy`.
5. Enter the network **Chain ID**.
6. Enter the authorization token that is required by the RPC proxy.

7. Optionally, set **Gas Price Wei**, **Allow Self Signed**, and a **CA Certificate Path** for a private certificate authority. Leave **BESU Accounts** empty unless your deployment model specifically requires non-custodial signing. The Oracle Blockchain Platform wallet service deployments collect signing details by deployment.
8. Select **Create**.

For the stored Oracle Blockchain Platform host, the extension derives the Besu proxy endpoint and uses the Oracle Blockchain Platform governance intent endpoint. This keeps the environment portable between deployment, execution, and governance operations.

The built-in **Hardhat Local Node** environment is available for local development and does not need to be saved. It is not an Oracle Blockchain Platform environment.

Deploy to an Oracle Blockchain Platform Besu Environment

1. In the **Contracts** pane, open the project's **Deploy / Execute** form and select the **Deploy** tab.
2. Select the contract package and the Besu environment.
3. Enter a deployment ID, or keep the displayed deployment ID.
4. Complete the required registry metadata. The deployment-time token name, symbol, description, URI, and administrator IDs are optional. Empty values use the deployment script defaults.
5. Enter the wallet service signing information shown for the Oracle Blockchain Platform environment.
 - **Wallet Service Sign URL**
 - **Wallet Service Accounts**, a JSON array of address/wallet-ID objects
 - **Wallet Service Auth Token**
6. Select **Next**.
7. In **Deployment Options**, enter the **Governance Contract Address** and **Policy UUID**, then select **Deploy**.

Blockchain App Builder runs the generated Hardhat deployment script through the Oracle Blockchain Platform Besu network, using the wallet service values only for that deployment. It deploys or reuses the required components, sets their contexts, and returns the token and account proxy addresses. Both proxies are inactive.

Submit the Deployment Intent

After the inactive deployment completes, the governance section is populated with the deployed **Token Target** and **Account Target** values. Keep both targets in the intent. Each is an EIP-1967 proxy; governance activates the proxy against its resolved implementation address and runtime code hash.

1. Confirm the token and account targets. Override them only if you are intentionally proposing an existing proxy pair.
2. Enter a future **Proposal Deadline**.
3. For Oracle Blockchain Platform Besu, enter the **From Address** authorized by the selected policy. The selected environment must have an RPC URL host and authorization token.
4. For other EVM environments, select a **Caller Account** or enter the caller's private key. The application derives the sender address from that key.

5. Select **Review**. The extension resolves each proxy's implementation and implementation code hash, and then displays the proposed intent.
6. Verify the UUID, deadline, proxy targets, implementation addresses, and code hashes.
7. Select **Submit**.

For an Oracle Blockchain Platform environment, the extension submits the reviewed intent through the Oracle Blockchain Platform wallet service governance endpoint. For other EVM environments, it sends a signed `proposeDeployIntent` transaction directly to the governance contract via JSON-RPC.

If you change the governance address, UUID, proxy targets, or deadline after review, the preview is invalidated. Review the proposal again before submitting it.

Governance Activation

The same intent workflow is used for both governance types. The difference is what happens after the governance contract accepts the proposal:

- **No-Op:** A successful `proposeDeployIntent` call activates the included components immediately. Confirm the transaction or Oracle Blockchain Platform response, and then verify the token's active state.
- **Governed:** The submitted intent remains subject to the configured policy. The token and account become active only after the required approvals have been received before the deadline.

The extension submits the intent; it does not impersonate or bypass governance approvers. For a governed policy, use the organization's approval process and wait for the governance contract to activate. If the policy rejects or expires the intent, correct the cause and then submit a new valid intent. Deployment alone will not activate the proxies.

Note

After you deploy a contract, manifest files are generated in the `.openzeppelin` folder under the project root. For production networks (not development networks), commit these files to source control. For more information, see [Network Files](#) in the OpenZeppelin documentation.

Troubleshooting Deployment Errors

Condition	Resolution
Oracle Blockchain Platform environment cannot be saved	Enter an HTTP or HTTPS origin without an endpoint path. The Oracle Blockchain Platform field expects the host only.
Governance review says that the Oracle Blockchain Platform environment needs an RPC URL host and authorization token	Update the saved Oracle Blockchain Platform environment with both values, and then open or select it again.
Governance review reports missing runtime bytecode	Verify the selected network and proxy/implementation addresses. Do not submit an intent against a target that is not deployed.
Deployment succeeds but the token is inactive	Submit the deployment intent. For governed governance, wait for all policy approvals; for no-op governance, confirm that the proposal transaction succeeded.

Condition	Resolution
Proposal cannot be submitted after changing fields	Select Review again. The application deliberately invalidates a stale preview.

Run a Contract

You can use Blockchain App Builder to query a deployed token or account/identity proxy, or to invoke a state-changing contract function.

After you compile a project, you can use the **Execute** tab in the **Deploy / Execute** form to query or invoke functions. The extension reads the selected contract ABI, presents the available functions, encodes the parameters, and sends the request through the transport appropriate for the selected environment. For composed projects, deploy and activate the token and account proxies before you attempt state-changing token operations.

The Execute tool discovers functions from compiled artifacts. For composed projects, the generated manifest also identifies the compatible account/identity ABI for the selected token. To access the Execute tool:

1. Open the **Contracts** view.
2. Select the Hardhat project and then open **Deploy / Execute**.
3. Select the **Execute** tab.
4. If the form includes a project selector, choose the project that owns the contract and its compiled artifacts.

You must complete the following fields to select a target and function before running a call.

- **Token:** The compiled token contract whose ABI you want to use.
- **Call Target:** Select **Token** to call the token proxy, or **Account / Identity** to call the paired account proxy. Select **Token** as the call target for token behavior such as balances, minting, transfers, configuration, and asset-class operations. Select **Account / Identity** for the companion account contract's identity, permission, or policy functions.
- **Target ABI:** The ABI used to list and encode functions. A generated manifest locks the paired account ABI; otherwise select one manually.
- **toAddress:** The deployed proxy address for the selected target. Do not enter a helper-library or implementation address.
- **Environment:** The environment where that proxy is deployed.
- **Function:** The ABI function to query or invoke. The list includes the function origin to help distinguish inherited capabilities.

Execution Path

The extension chooses or restricts the available mode based on the environment that you select.

Environment and API Mode	Query Behavior	Invocation Behavior	Required Signer Information
Hardhat Local Node or Other EVM with JSON-RPC	Calls the node using the selected local private key as <code>from</code> .	Signs and broadcasts directly through JSON-RPC.	A configured caller account or manually entered 32-byte private key.

Environment and API Mode	Query Behavior	Invocation Behavior	Required Signer Information
Oracle Blockchain Platform Besu with JSON-RPC APIs	Calls the Oracle Blockchain Platform JSON-RPC endpoint using the selected Wallet Service account address as from.	Builds a transaction, requests a wallet service signature, then broadcasts it through Oracle Blockchain Platform JSON-RPC.	Wallet service sign URL, accounts JSON, authorization token, and selected wallet account.
Oracle Blockchain Platform Besu with REST rpc-proxy APIs	Posts to the Oracle Blockchain Platform query API.	Posts to the Oracle Blockchain Platform transaction API.	From Address; optionally gas and receipt-wait settings. No local private key or Wallet Service signer is entered in this mode.

REST rpc-proxy mode is available only for an Oracle Blockchain Platform Besu environment. The environment's RPC URL Host, authentication token, and TLS settings are used for the REST calls. Other EVM and local Hardhat environments use JSON-RPC with local signing.

Query a Function

Use **Query** for a read call or to simulate a function against the current chain state.

1. Select the target, environment, function, and parameters.
2. Configure the caller or wallet information required by the selected execution path.
3. Select **Query**.
4. Review the **Output** field. JSON-RPC output includes the raw call result and decoded ABI result when decoding succeeds. REST output includes the API response.

Queries use a call request; they do not broadcast a transaction or persist state. A query can still revert if the selected function's conditions, caller permissions, or supplied parameters are not valid for the current contract state.

Invoke a Function

Use **Invoke** for a state-changing transaction.

1. Select the target, environment, function, and parameters.
2. Configure the required signing method.
 - For **Other EVM** or **Hardhat Local Node**, select a configured caller account or enter the authorized private key.
 - For **Oracle Blockchain Platform Wallet Service**, supply the Sign URL, accounts JSON, authorization token, and select the wallet account.
 - For **REST rpc-proxy**, enter the From Address and, when needed, gas limit and receipt wait settings.
3. Select **Invoke**.
4. Wait for the result in **Output**. Successful JSON-RPC and Wallet Service invokes include transaction hash, block number, gas used, sender, target, and receipt status.

An invocation can be accepted by the transport but still revert on-chain. Treat a receipt status other than 1, or an error response, as a failed operation and inspect the output for a decoded custom error or RPC error details.


```
}
]
```

Select one of the parsed accounts in **Wallet Account**. Queries use its address as from; invocations ask the wallet service to sign for that account. The wallet service authorization token is not stored in the form's saved draft.

Save and Reuse Inputs

Select **Save Input** to retain the current non-secret Execute selections and parameter JSON for the project. Select **Clear Saved Input** to remove the saved form state. Recheck proxy addresses, selected function, and parameters before an invocation; saved inputs do not prove that a target is still deployed on the currently selected environment.

Troubleshooting

Condition	Resolution
After an upgrade, a newly added method is not listed in the ABI list on the Execute page	Update the Blockchain App Builder manifest file (<code>generated.manifest.json</code>) to change <code>contractClassName</code> and <code>contractRelativePath</code> to point to the newly upgraded contract.
No ABI functions found	Compile the project and confirm that the selected token/account artifact exists. For an account target without a manifest, select the correct account ABI manually.
Target address has no contract code	Check the selected environment and enter the deployed proxy address, not a helper library or implementation address.
Invalid parameters JSON	Use the ... editor to regenerate the argument array. Ensure arrays and objects use valid JSON with double-quoted property names.
Wallet service account selection is required	Enter a valid accounts JSON array and select a Wallet Account .
REST rpc-proxy APIs require an Oracle Blockchain Platform environment	Switch to an Oracle Blockchain Platform Besu environment with a host-only RPC URL, or use JSON-RPC for Other EVM/Hardhat.
Unauthorized caller or a decoded custom error	Verify the selected caller/wallet account, target proxy, active governance state, role/policy prerequisites, and ABI parameters.
Query or Invoke remains unavailable	Complete the required target, environment, function, and signer fields for the selected API mode.

Debug a Contract

You can use Solidity console logs to debug contracts that are running on the simulated Hardhat local node.

To debug a contract, deploy it to the built-in **Hardhat Local Node** environment, and make all logging changes only in a local debugging branch or temporary working copy.

The built-in local environment is named `hardhatnode` and uses `http://127.0.0.1:8545` with chain ID 31337. It is not a saved Oracle Blockchain Platform or other EVM environment. When you select it to deploy or run a contract and no local node is listening, Blockchain App

Builder for Besu starts a Hardhat node for the selected project and opens the Hardhat local node terminal.

1. At the top of the Solidity file that contains the code path that you want to inspect, add the Hardhat console import.

```
import "hardhat/console.sol";
```

2. Add a temporary log in the function or internal helper to investigate.

```
console.log("value", value);
```

The following code shows common log examples.

```
console.log("entered transfer");
console.log("caller", msg.sender);
console.log("amount", amount);
console.log("is allowed", isAllowed);
```

Use the value type that matches the variable that you are inspecting. These examples cover strings, addresses, integer values, and Boolean values. Keep log statements narrow and temporary; do not log private keys, authentication tokens, or other credentials.

3. In the **Deploy / Execute** form, select **Clean** if you need to discard old artifacts, and then select **Compile**.
4. Select **Hardhat Local Node** as the environment.
5. Deploy the local contract instance, or use a local proxy that is already deployed. Satisfy any project-specific deployment prerequisites, such as governance contracts, in the local network.
6. Open the **Execute** pane, select the local token or account proxy, enter the function parameters, and then select **Query** or **Invoke**.
Query completes a local call and **Invoke** sends a local transaction. Either can reach a logged code path, but only an invocation changes the simulated chain state.
7. Open the **Hardhat Local Node** terminal in Visual Studio Code and inspect the emitted log lines.
8. Remove the `hardhat/console.sol` import statement and every temporary `console.log` statement.
9. Clean and compile again before committing code or using an Oracle Blockchain Platform Besu, other EVM, or production environment.

Do not attempt to use the Hardhat-only instrumentation for Oracle Blockchain Platform or other EVM environments. Instead, use ordinary application events, off-chain transaction tracing, or environment-approved observability. Console logging is not a replacement for production diagnostics.

Troubleshooting the Debugging Process

Condition	Resolution
<code>hardhat/console.sol</code> cannot be found	Run Install Dependencies in the selected project, and then compile again. The import is provided by the project's Hardhat dependency.

Condition	Resolution
The local node does not start	Confirm that the selected project contains <code>hardhat.config.ts</code> or <code>hardhat.config.js</code> , verify Node.js and npm are available to Visual Studio Code, and check the Hardhat Local Node terminal for startup errors.
No logs are displayed	Confirm that Hardhat Local Node is the selected environment, that the invoked code path reaches the log statement, and that you are viewing the Hardhat Local Node terminal rather than an Oracle Blockchain Platform or project terminal.
The application is using old code	Select Clean , then Compile , before deploying or invoking again.
A non-local build or deployment fails after debugging	Remove every <code>hardhat/console.sol</code> import and <code>console.log</code> call, then clean and compile the project again.

Upload a Contract Manifest to the Registry

You can use Blockchain App Builder to validate and publish deployment metadata from a selected Hardhat project to contract registry services.

The **Upload Contract Manifest** tool runs the project-owned `hardhat-upload-contract-artefacts` plug-in and passes registry endpoints as runtime values; it does not modify the project's Hardhat variables.

Manifest upload is available only for saved Oracle Blockchain Platform Besu environments.

Prerequisites

- Select an imported or composed Hardhat project that is deployed successfully.
- Confirm that the project includes the `hardhat-upload-contract-artefacts` package in the `package.json` file, imports the plug-in in `hardhat.config.ts` or `hardhat.config.js`, and defines the `uploadContractManifest` configuration.
- Create or select an Oracle Blockchain Platform Besu environment with the correct chain ID and RPC proxy authentication.
- Ensure that the required local deployment records exist for the upload mode that you choose.

The following upload modes are supported.

Upload Mode	Required Local Records	Registry Destination
Both	OpenZeppelin manifest, ERC-7201 deployed-contracts JSON, and SOLC deployed-contracts JSON	Both registry endpoints
ERC-7201 / Upgradeable	OpenZeppelin manifest and ERC-7201 deployed-contracts JSON	ERC-7201 registry endpoint
Immutable / SOLC	SOLC deployed-contracts JSON	SOLC registry endpoint

For a prepackaged imported project or digital assets contracts generated by Blockchain App Builder, typically ERC-7201 mode is sufficient.

For chain ID 1500, the default paths are as follows.

```
./openzeppelin/unknown-1500.json
./obp-da/.erc7201/deployedContracts-1500.json
./obp-da/.solc/deployedContracts-1500.json
```

Adjust the paths if the project stores deployment records elsewhere.

1. Open **Developer Tools** and then select **Upload Contract Manifest**.
2. Select the Hardhat project and an Oracle Blockchain Platform Besu environment.
3. Select the upload mode.
4. Enter the required registry endpoint URL or URLs:
 - **ERC-7201 Manifest Upload Registry Endpoint** for upgradeable records
 - **SOLC Manifest Upload Registry Endpoint** for immutable records
5. Confirm the OpenZeppelin and deployed-contract file paths. Optionally set timeout and verbose output.
6. Select **Validate**. The plug-in runs and does not publish registry data.
7. When validation succeeds, clear **Validate** and select **Upload**.

Troubleshooting

Condition	Resolution
No environment is available	Create a saved Oracle Blockchain Platform Besu environment. EVMs and the Hardhat local node are not valid environments for uploading manifests.
Registry endpoint is invalid	Enter a complete HTTP or HTTPS registry URL for every endpoint required by the selected mode.
Required file not found	Compile and deploy the relevant contracts or correct the project-relative manifest and file path of the deployed contract.
Plug-in or configuration is missing	Install or import hardhat-upload-contract-artefacts and add the uploadContractManifest configuration that is shown in the tool's preflight guidance.
Upload fails after dry run succeeds	Review registry-service authentication, availability, endpoint policy, and the full plug-in output. The dry run validates local inputs but does not guarantee registry acceptance.

Generate a Wrapper API Specification and Postman Collection

You can use Blockchain App Builder to create filtered token and account/identity endpoint specification files and a matching Postman collection.

Confirm that the package.json file includes hardhat-plugin-obp-besu-wrapper-api and that the Hardhat configuration imports it. Select a generated token package with a manifest and compiled token/account artifacts. The tool discovers composed packages from their generated manifest and also supports compatible imported reference-project manifests. It does

not deploy or start a wrapper API service. The wrapper API service is already run as part of instance setup.

1. Generate token and account specifications.
 - a. Open **Developer Tools** and select **Generate Wrapper API Specification and Postman Collection**.
 - b. In **Configuration**, select the Hardhat project, token package, and an output folder inside the project.
 - c. In **Mapping**, review the token APIs and account/identity APIs. Keep **Generate Wrapper APIs** selected for each method that will remain in the endpoint specification. Edit the endpoint path where required.
 - d. Select **Generate** to write the filtered token and account specification JSON files and the Postman collection.
2. Add ORDS Swagger/OpenAPI endpoints.
 - a. In the **Mapping** view, selected **ORDS API**. The tool supports ORDS API input only.
 - b. Select an ORDS Swagger/OpenAPI JSON file.
 - c. Generate the wrapper assets again.
3. Create a combined API, which defines a sequential workflow that combines selected token or account methods into one wrapper API endpoint.
 - a. In **Mapping**, select **Use in Combined API** for each eligible method.
 - b. Select **Create Combined API**.
 - c. Enter the combined API name and endpoint path.
 - d. Review and reorder the selected steps. For each step, map its request-payload source fields.
 - e. Save the combined API draft, then select **Generate**.
4. Register specifications with a wrapper API service.
 - a. Ensure that a wrapper API service is running and select the **Deploy** tab.
 - b. Select a generated token specification, account specification, both specifications, or provide a custom JSON body.
 - c. Enter the wrapper API host URL.
 - d. Select an Oracle Blockchain Platform Besu environment so the request can reuse its authentication and TLS settings.
 - e. Review the registration preview and register the selected specifications.

Registration posts to the wrapper API service's /register endpoint. The tool can use a selected Oracle Blockchain Platform environment's private CA path or explicit self-signed setting for the registration request; it does not deploy the service itself.

Troubleshooting

Condition	Resolution
Artifacts or manifest are missing	Compile the selected project and confirm that the generated/imported package files exist.
No endpoints are generated	Select at least one method for wrapper API generation, or save at least one valid combined API.

Condition	Resolution
ORDS generation fails	Provide an ORDS Swagger/OpenAPI JSON file and confirm that the project plug-in supports the <code>--ords-swagger</code> option. Inspect the full plug-in log for schema or operation errors.
Combined API is rejected	Check the API name/path, ordered steps, and request-field mappings, then save the draft again.
Registration is unavailable	Generate endpoint specification files first, then select an Oracle Blockchain Platform Besu environment and provide a reachable wrapper API host URL.

Generate an RPC Proxy Postman Collection

You can automatically create a Postman collection for a deployed package's token and account/identity ABIs.

The generated collection targets the Oracle Blockchain Platform RPC proxy API and is intended for testing contract calls outside of the Visual Studio Code **Execute** form.

Confirm that the project imports the `generate-rpc-proxy-postman` Hardhat task in `hardhat.config.ts` or `hardhat.config.js`. The generation tool does not deploy contracts or create a saved environment. It generates a Postman collection from ABI artifacts. You must provide the deployed proxy addresses and environment variables in Postman when you run requests.

1. Open **Developer Tools** in the Blockchain App Builder For Besu activity bar.
2. Select **Generate RPC Proxy Postman Collection**.
3. Select the Hardhat project.
4. Select the token package. Composed packages and supported imported packages resolve their paired token and account/identity artifacts. Review the token and account artifact paths to ensure that they are correct.
5. Enter an output path, or retain the following default path:

`generated/rpc-proxy-postman/<token-name>-rpc-proxy.postman.collection.json`

6. Select **Generate**. The status area reports the output location and exposes the full Hardhat log if collection generation fails or produces warnings.

To use the generated collection, import the JSON output file into Postman. Configure the collection variables that are required by the generated requests, such as the Oracle Blockchain Platform RPC proxy host, authentication token, deployed token/account proxy addresses, and any caller or wallet information that is required by the endpoint.

The collection represents the ABI surface at the time of compilation. Regenerate it whenever token/account sources, generated composition, or compiled artifacts change.

Troubleshooting Postman Collection Generation

Condition	Resolution
Token or account artifact is required	Compile the selected project, and then open or select the token package again.

Condition	Resolution
The project does not include generate-rpc-proxy-postman	Import the project task in hardhat.config.ts, as showing in the following example. <pre>import "../tasks/generate-rpc-proxy-postman";</pre>
Output file is missing	Check the selected project root and the relative output path in the full log.
Postman calls an unexpected method set	Recompile the current project and regenerate the collection from the intended token package.

Upgrade the Blockchain App Builder Extension for Visual Studio Code

To upgrade Blockchain App Builder, you must uninstall the previous version and then install the newer version.

1. In Visual Studio Code, open the Extensions panel and then select the Blockchain App Builder extension. The Blockchain App Builder extension page is displayed.
2. Click **Uninstall**.
3. After Blockchain App Builder is uninstalled, restart Visual Studio Code.
4. Download the latest version of the Blockchain App Builder extension from the Developer Tools tab on the Blockchain App Builder pane of the Oracle Blockchain Platform console, and then install Blockchain App Builder.

3

Using the Digital Assets Hardhat Command-Line Interface

You can use the command-line interface (CLI) to install, compile, deploy, and run smart contracts.

The digital assets Hardhat CLI is available as part of any digital assets project generated by using Blockchain App Builder with Oracle Blockchain Platform Enterprise Edition for Besu. The CLI is also included in the supplied reference projects for wholesale CBDC, stablecoins, loyalty tokens, tokenized deposits, bond marketplaces and art loyalty tokens. You can use the command-line interface with digital assets projects to compile, deploy, upgrade, and upload contracts to the registry. You can also generate wrapper API specifications and Postman collections. You can run and test digital assets projects on a local Besu network, an in-memory evm node for development testing, or a provisioned Oracle Blockchain Platform Besu network.

Table 3-1 Workflow When Using the Visual Studio Code Extension

Task	Description	More Information
Install and configure	Download a packaged sample smart contract from your Oracle Blockchain Platform console and install it and any prerequisite software.	Install the Digital Assets Hardhat Command-Line Interface
Compile the smart contract	Compile the digital assets contract so that the bytecode and application binary interface (ABI) are available for deployment.	Compile a Contract with the CLI
Deploy the smart contract	Deploy the digital assets project locally to the included preconfigured Besu network, local evm node, or remotely to Oracle Blockchain Platform Enterprise Edition for Besu.	Deploy a Contract with the Command Line
Upgrade the smart contract	Upgrade the contract to a new version and test it in any of the deployment environments.	Upgrade a Contract
Upload the contract manifest to the registry	Upload the generated Besu contract manifest to the Oracle Blockchain Platform registry service to capture the ABI, bytecode, and storage layout.	Upload a Contract Manifest to the Registry with the Command Line
Generate a wrapper API specification and Postman collection	Generate the wrapper API specification and Postman collection, which contain API endpoints for each contract method. You can also generate a Postman collection for the RPC proxy calls.	Generate a Wrapper API Specification and Postman Collection with the Command Line
Generate an RPC proxy API Postman collection	Generate the Postman collection for RPC proxy calls	Generate an RPC Proxy Postman Collection with the Command Line

Install the Digital Assets Hardhat Command-Line Interface

You can download the Hardhat CLI tool from the Oracle Blockchain Platform console.

Prerequisites

Before you install the Hardhat CLI on your local system, you must install the prerequisites.

- Node.js version 22.17.0 or later
- npm version 10.9.2 or later

Use nvm to install Node.js and npm. For more information, see [Node Version Manager](#) at GitHub.

Install the Hardhat CLI

1. Download the packaged smart contract application that you want to use from the ERC20+ or ERC1155+ areas in the Oracle Blockchain Platform interface.
2. Run the following command to extract the package.

```
unzip <downloaded-pre-packaged-name>.zip -d <path/to/directory>
```

3. Navigate to the directory where you extracted the package.

```
cd <path/to/directory>
```

4. Run the following commands to install the dependencies.

```
npm install
```

5. Run the following command to confirm that version 2.28.6 of Hardhat is installed.

```
npx hardhat -version
```

Compile a Contract with the CLI

After you extract a packaged contract, you can compile the project from the command line.

Run the following commands to compile and verify a contract.

1. Navigate to the directory that contains the extracted package.

```
cd <path/to/directory>
```

2. Compile the project.

```
npx hardhat clean && npx hardhat
```

3. Navigate to the contracts folder.

```
cd <path/to/directory>/artifacts/contracts
```

Verify that the ABI, bytecode, and other compilation related information is available under the contracts folder. For example, in case of the TokenizedDeposit package, the information will be found in the `./artifacts/contracts/TokenizedDeposit/TokenizedDepositUpgradeable.sol/TokenizedDepositUpgradeable.json` file.

Deploy a Contract with the Command Line

After you compile a smart contract you can deploy it to an Oracle Blockchain Platform Besu environment.

The following steps walk through how to deploy the packaged Tokenized Deposit contract project. For other packaged contracts, see the README file that is bundled with the contract package.

Configure the Oracle Blockchain Platform Besu Network

Run the following commands to set the values that are consumed by the project's `hardhat.config.ts` file.

```
npx hardhat vars set BESU_CHAIN_ID <chain-id>
npx hardhat vars set BESU_RPC_URL <rpc-proxy-url>
npx hardhat vars set BESU_RPC_PROXY_AUTH_TOKEN "Bearer <rpc-auth-token>"
npx hardhat vars set WALLET_SERVICE_BASE_URL <wallet-service-sign-url>
npx hardhat vars set WALLET_SERVICE_ENABLED true
npx hardhat vars set WALLET_SERVICE_ACCOUNTS
'[{ "address": "0x<address>", "walletId": "0x<address>" } ]'
npx hardhat vars set WALLET_SERVICE_AUTH_TOKEN "Bearer <wallet-service-auth-token>"
```

For the Tokenized Deposit project, the `BESU_RPC_URL` value is the complete configured JSON-RPC proxy endpoint, as shown in the following example.

```
https://rpcproxy.example.oracle.com/v1/besu/proxy
```

The `WALLET_SERVICE_BASE_URL` value is the complete `/transaction` endpoint, as shown in the following example.

```
https://rpcproxy.example.oracle.com/v1/besu/transaction
```

Deploy the Inactive Proxy Pair

Run the ERC-5982 deployment script with a stable deployment ID, as shown in the following example. The script deploys or resumes the ERC-20 helper libraries, account proxy, and Tokenized Deposit proxy. It also links the account/token contexts and sets governance context on both proxies.

```
DEPLOYMENT_ID=<deployment-id> \
GOVERNANCE_ADDRESS=<governance-contract-address> \
GOVERNANCE_UUID=<governance-uuid> \
TOKEN_NAME="DepositToken" \
TOKEN_SYMBOL="DP" \
TOKEN_DESCRIPTION="DepositToken proxy contract" \
ADMIN_USER_ID="platformadmin1" \
ADMIN_ORG_ID="org1" \
```

```

CONTRACT_METADATA_SMART_CONTRACT_ID="DP123" \
CONTRACT_METADATA_VERSION="1.0.0" \
CONTRACT_METADATA_DESCRIPTION="DepositToken proxy contract" \
CONTRACT_METADATA_AUTHOR="Oracle" \
CONTRACT_METADATA_TAGS='{"assetType":"tokenized deposit","role":"proxy"}' \
npx hardhat run scripts/deploy/deposittoken/deploy-deposittoken-erc-5982.ts --
network obp-besu

```

The `DEPLOYMENT_ID`, `GOVERNANCE_ADDRESS`, and `GOVERNANCE_UUID` values are required. Reuse the same deployment ID only to resume an interrupted deployment. After a deployment completes successfully, its resume cache is deleted; rerunning with that ID starts a fresh deployment.

The `CONTRACT_METADATA_*` values describe the Tokenized Deposit proxy in the ERC-7201 deployment registry. `CONTRACT_METADATA_TAGS` must be a JSON object whose values are strings.

The command completes with the token and account proxies inactive. Preserve the printed `DEPLOYMENT_SUMMARY_JSON` and record both proxy addresses; they are required for the governance intent.

Submit the Deployment Intent

In the following script, set a future deadline by editing the `DEADLINE_SECONDS` value if the default one-hour proposal window is not appropriate for your environment. Include every proxy whose governance context was set by the deployment. For the Tokenized Deposit package, typically this means both the token proxy and its account proxy. For an Oracle Blockchain Platform Besu proposal submitted via the RPC proxy and wallet service route, run the following script.

```

GOVERNANCE_INTENT_MODE=obp-besu \
GOVERNANCE_ADDRESS=<governance-contract-address> \
GOVERNANCE_UUID=<governance-uuid> \
DEPLOY_TARGETS_JSON='["<token-proxy>","<account-proxy>"]' \
DEADLINE_SECONDS=3600 \
DEPLOY_INTENT_MANIFEST=.obp-da/governance/deploy-intent.json \
npx hardhat run scripts/deploy/deposittoken/governance/propose-deploy-
intent.ts --network obp-besu

```

The script validates each target, resolves its EIP-1967 implementation and runtime code hash, submits the proposal, and writes the reviewed components and proposal result to the file specified by the `DEPLOY_INTENT_MANIFEST` variable.

For a direct JSON-RPC governance submission, use a Hardhat network with a local signer and change only the proposal transport and network, as shown in the following example.

```

GOVERNANCE_INTENT_MODE=json-rpc \
GOVERNANCE_ADDRESS=<governance-contract-address> \
GOVERNANCE_UUID=<governance-uuid> \
DEPLOY_TARGETS_JSON='["<token-proxy>","<account-proxy>"]' \
DEADLINE_SECONDS=3600 \
DEPLOY_INTENT_MANIFEST=.obp-da/governance/deploy-intent.json \
npx hardhat run scripts/deploy/deposittoken/governance/propose-deploy-
intent.ts --network <direct-json-rpc-network>

```

The direct network must be configured with an authorized non-custodial signer. Do not use JSON-RPC with a wallet-service-only configuration.

Governance Activation

Submitting deployment intent is a separate process from deploying the proxies.

- **No-Op:** The included components activate during successful `proposeDeployIntent` processing. Verify the transaction or Oracle Blockchain Platform response and then query the active state.
- **Governed:** The intent remains pending until the configured policy receives the required approvals before its deadline. The token and account activate automatically only after that policy completes.

The CLI does not bypass governance approvers. If the proposal expires, is rejected, or uses the wrong proxy/code hash, resolve the cause and submit a new intent rather than treating the initial deployment as active.

After governance is activated, verify the deployment. Use the generated Postman or wrapper API assets, or import the project into Blockchain App Builder for Besu and use the **Execute** panel. Use the proxy addresses from the deployment summary as call targets; do not use helper-library or implementation addresses.

Note

After you deploy a contract, manifest files are generated in the `.openzeppelin` folder under the project root. For production networks (not development networks), commit these files to source control. For more information, see [Network Files](#) in the OpenZeppelin documentation.

Troubleshooting

Message or Condition	Resolution
GOVERNANCE_ADDRESS and GOVERNANCE_UUID are required	Provide both values to the deploy command and verify the governance address has bytecode on the selected network.
Deployment is incomplete	Re-run the same command with the same DEPLOYMENT_ID value before the deployment completes.
DEPLOY_TARGETS_JSON is invalid	Pass a non-empty JSON array of unique proxy addresses, for example <code>["0x...token", "0x...account"]</code> .
Target is not an EIP-1967 proxy	Use the deployed token and account proxy addresses from the DEPLOYMENT_SUMMARY_JSON file, not an implementation or library address.
Oracle Blockchain Platform intent submission fails because authorization is absent	Confirm that the BESU_RPX_PROXY_AUTH_TOKEN value is set and that the configured Oracle Blockchain Platform network carries it as an Authorization header.
Deployment succeeds but the token remains inactive	Submit the deployment intent. For governed governance, wait for policy approvals; for No-Op governance, confirm that proposal processing succeeded.

Upgrade a Contract

You can upgrade a packaged contract from the command line.

The following steps show how to upgrade the sample Tokenized Deposit contract project. For other packaged contracts, see the README file bundled with the contract package.

An upgrade changes the implementation behind the existing proxy; it does not create a new proxy or reset its storage.

The Hardhat command-line interface supports upgrading contracts. If you deployed a smart contract using the RPC proxy API, it is immutable and cannot be upgraded.

The reference project includes V2 implementations for demonstration and testing:

- Token: `contracts/TokenizedDeposit/_testContracts/TokenizedDepositUpgradeableV2.sol`
- Account: `contracts/obp-sdk/accounts/modules/v1/_testContracts/AccountWithERC5982UUPSv2.sol`

The V2 contracts preserve the V1 storage layout, expose `version()` as `v2.0.0`, and include an empty `initializeV2()` reinitializer. Treat them as reference code; review and replace them with your own compatible implementation for an actual upgrade.

To upgrade a contract you need the proxy address for the contract, a configured Oracle Blockchain Platform Besu network, and authority to submit the governance proposal. The deployed contract must be active and eligible for its configured upgrade authorization path.

Before you modify an implementation, preserve storage compatibility. Append storage fields only. Do not reorder, remove, or change the type of existing fields, alter inherited storage layout, or bypass the upgrade validation. If the change is incompatible, deploy a new proxy and explicitly migrate the state instead.

① Note

You must point to the upgraded source. The `IMPLEMENTATION_CONTRACT` variable is passed to Hardhat's contract factory. Set it to the fully-qualified Solidity contract name for the upgraded version.

```
<path-to-upgraded-contract>.sol:<upgraded-contract-name>
```

Do not point the variable at the current V1 contract. For example, the Tokenized Deposit V2 reference implementation is shown in the following example.

```
contracts/TokenizedDeposit/_testContracts/  
TokenizedDepositUpgradeableV2.sol:TokenizedDepositUpgradeableV2
```

If you create a different V3 or custom implementation, replace both the path and contract name in the command with that upgraded source. Compile after the change and before running the prepare command.

Upgrade the Token Proxy

1. Run the following command to prepare the token upgrade. Preparation validates the implementation and storage layout, deploys the exact new implementation, records its code hash and library addresses in an upgrade manifest, and submits the required governance intent. It does not upgrade the proxy yet.

```
UPGRADE_ACTION=prepare \
GOVERNANCE_INTENT_MODE=obp-besu \
TOKEN_PROXY=<token-proxy-address> \
IMPLEMENTATION_CONTRACT='contracts/TokenizedDeposit/_testContracts/
TokenizedDepositUpgradeableV2.sol:TokenizedDepositUpgradeableV2' \
LIBRARIES_JSON='{"ERC20TokenHelperLib":null,"ERC20MultiLevelApprovalHelperLib":null}' \
DEPLOY_LIBRARIES=true \
INITIALIZER_DATA=0x \
UPGRADE_MANIFEST=.obp-da/upgrades/deposittoken-v2.json \
npx hardhat run scripts/upgrade/upgrade-deposittoken.ts --network obp-besu
```

With the `DEPLOY_LIBRARIES` set to `true`, a null library entry is deployed and its address becomes part of the reviewed implementation code hash. To reuse libraries, supply existing compatible library addresses in the `LIBRARIES_JSON` variable. The `INITIALIZER_DATA` variable is raw ABI-encoded calldata passed to the `upgradeToAndCall` function. Use `0x` for the supplied reference V2 contracts because their `initializeV2()` is empty. If your new implementation needs to be initialized, provide correctly encoded reinitializer calldata and review it as part of the upgrade proposal.

2. Approve the governance proposal. If the `GOVERNANCE_INTENT_MODE` variable is set to `obp-besu`, the prepare step submits the proposal through the Oracle Blockchain Platform RPC proxy. Obtain the approvals required by the target governance configuration. The proxy remains on its old implementation until the proposal is approved and the execute step completes. Use `GOVERNANCE_INTENT_MODE=json-rpc` only when the project is configured to submit the governance intent directly to the contract instead of through the Oracle Blockchain Platform endpoint.
3. Run the token upgrade. After approval, run the same manifest that the prepare command wrote.

```
UPGRADE_ACTION=execute \
UPGRADE_MANIFEST=.obp-da/upgrades/deposittoken-v2.json \
npx hardhat run scripts/upgrade/upgrade-deposittoken.ts --network obp-besu
```

The script verifies the manifest chain ID, deadline, and prepared implementation bytecode hash before calling the `upgradeToAndCall` function.

Upgrade the Account Proxy

1. Run the following command to prepare the account upgrade.

```
UPGRADE_ACTION=prepare \
GOVERNANCE_INTENT_MODE=obp-besu \
ACCOUNT_PROXY=<account-proxy-address> \
IMPLEMENTATION_CONTRACT='contracts/obp-sdk/accounts/modules/v1/
_testContracts/AccountWithERC5982UUPSv2.sol:AccountWithERC5982UUPSv2' \
LIBRARIES_JSON='{}' \
```

```
DEPLOY_LIBRARIES=false \
INITIALIZER_DATA=0x \
UPGRADE_MANIFEST=.obp-da/upgrades/deposittoken-account-v2.json \
npx hardhat run scripts/upgrade/upgrade-account.ts --network obp-besu
```

Replace the fully-qualified value of the `IMPLEMENTATION_CONTRACT` variable with the path and contract name of your new account implementation. Do not reuse the token's source path, proxy address, or manifest for an account upgrade.

2. After the Account upgrade proposal receives the required governance approvals, run the following command to approve and complete the account upgrade.

```
UPGRADE_ACTION=execute \
UPGRADE_MANIFEST=.obp-da/upgrades/deposittoken-account-v2.json \
npx hardhat run scripts/upgrade/upgrade-account.ts --network obp-besu
```

Troubleshooting

Condition	Resolution
Prepare fails storage validation	Do not force the upgrade. Correct the incompatible layout change, or deploy a new proxy and migrate the state.
A previous prepare manifest has expired or targets a different implementation	Create a replacement deliberately by setting <code>REPREPARE=true</code> and retaining the old manifest as an audit record.
Execute is run before approval	Obtain the governance approvals first. An approved intent is required before the prepared upgrade becomes executable.
The wrong implementation is selected	Verify that the <code>IMPLEMENTATION_CONTRACT</code> variable uses the upgraded source's fully qualified path:contract value, then recompile and prepare a new manifest. Never execute a manifest prepared for a different implementation.
Execute is rerun	The script detects a proxy that is already upgraded and records that outcome rather than sending a second upgrade transaction. For an older manifest that lacks <code>implementationContract</code> , supply the same <code>IMPLEMENTATION_CONTRACT</code> value when executing so that the implementation history can be synchronized.

Upload a Contract Manifest to the Registry with the Command Line

You can use the command line to validate and publish deployment metadata from a selected Hardhat project to contract registry services.

The `upload-contract-manifest` Hardhat task publishes the project's upgradeable or immutable contract manifests to configured registry services.

Run the task from the packaged contract project's root directory after a successful deployment. The task reads deployment manifest files from the project. The following files are the standard locations:

- OpenZeppelin manifest: `.openzeppelin/unknown-<chain-id>.json`
- ERC-7201 manifest: `.obp-da/.erc7201/deployedContracts-<chain-id>.json`
- Immutable/Solidity manifest: `.obp-da/.solc/deployedContracts-<chain-id>.json`

Use the chain ID of the target environment in place of `<chain-id>`. If your OpenZeppelin manifest is stored elsewhere, set the `OZ_MANIFEST_PATH` variable to its project-local path.

1. Configure registry endpoints. Set the registry endpoint variables for the manifest types that you plan to upload. Use your shell's secure environment-variable mechanism or the project's configured Hardhat variable mechanism; do not commit credentials or private endpoints to source control. For example, a one-time shell invocation can provide the values to the task, as shown in the following example.

```
ERC7201_MANIFEST_UPLOAD_REGISTRY_ENDPOINT="https://registry.example/erc7201" \
SOLC_UPLOAD_REGISTRY_ENDPOINT="https://registry.example/solc" \
npx hardhat upload-contract-manifest --mode all
```

For an Oracle Blockchain Platform environment, also provide the target chain information and the RPC proxy authorization token required by your project configuration.

```
BESU_CHAIN_ID="<chain-id>" \
BESU_RPX_PROXY_AUTH_TOKEN="<rpc-proxy-token>" \
ERC7201_MANIFEST_UPLOAD_REGISTRY_ENDPOINT="https://registry.example/erc7201" \
SOLC_UPLOAD_REGISTRY_ENDPOINT="https://registry.example/solc" \
npx hardhat upload-contract-manifest --mode all
```

The endpoint values are explicit configuration. Do not assume that a registry URL can be derived from an Oracle Blockchain Platform host name.

2. Select what to upload by choosing one of the following modes.

```
# Upload both applicable manifest types.
npx hardhat upload-contract-manifest --mode all

# Upload the OpenZeppelin and ERC-7201 manifest content.
npx hardhat upload-contract-manifest --mode erc7201

# Upload immutable/Solidity contract manifest content only.
npx hardhat upload-contract-manifest --mode immutable
```

The `all` and `erc7201` modes require the `ERC7201_MANIFEST_UPLOAD_REGISTRY_ENDPOINT` variable. The `all` and `immutable` modes require the `SOLC_UPLOAD_REGISTRY_ENDPOINT` variable. For packaged imported projects or digital assets contracts generated by Blockchain App Builder, ERC-7201 mode is sufficient.

3. Use the `--dry-run` option to validate the selected manifest files and upload configuration without publishing.

```
npx hardhat upload-contract-manifest --mode all --dry-run
```

4. After a successful dry run, run the same command without the `--dry-run` option to publish. If your project supports it, set the `UPLOAD_CONTRACT_MANIFEST_TIMEOUT_MS` variable to increase the upload timeout and set `UPLOAD_CONTRACT_MANIFEST_VERBOSE=true` for additional diagnostic output. Do not include secrets in verbose logs or shared terminal output.

Troubleshooting

Condition	Resolution
Task not found	Confirm that the manifest-upload plug-in is installed, imported by the <code>hardhat.config.ts</code> file, and configured with the upload task.
Required manifest is missing	Deploy to the selected environment first, confirm the chain ID, and check the expected manifest location. For a nonstandard OpenZeppelin manifest, set the <code>OZ_MANIFEST_PATH</code> variable to the correct project-local file.
Registry endpoint is missing or invalid	Provide an <code>http</code> or <code>https</code> endpoint for the mode that you selected. The <code>erc7201</code> and <code>immutable</code> modes use different endpoint variables.
Oracle Blockchain Platform upload is unauthorized	Verify that the selected Oracle Blockchain Platform environment, and <code>BESU_CHAIN_ID</code> and <code>BESU_RPX_PROXY_AUTH_TOKEN</code> variables are all for the same target environment.

Generate a Wrapper API Specification and Postman Collection with the Command Line

You can use Blockchain App Builder to create filtered token and account/identity endpoint specification files and a matching Postman collection.

The `wrapper-api-endpoints-config` Hardhat task generates endpoint configuration files and a Postman collection from deployed contract artifacts or ABI files. The Postman task can also incorporate an ORDS Swagger/OpenAPI document and combined API definitions.

Prerequisites

- Install the project's dependencies and compile if you plan to use artifacts.
- Ensure that the Wrapper API plug-in and its tasks are configured in the `hardhat.config.ts` file.
- Identify the token and, when applicable, account contract artifact or ABI files.

Generate an Endpoint Configuration

Run the following commands from the prepackaged contract project's root directory. Replace the example paths and names with paths in your project.

To generate an endpoint configuration file for each contract target that you want to expose, run the following commands.

```
npx hardhat generate:wrapper-api-endpoints-config \
  --artifact artifacts/contracts/Token.sol/Token.json \
  --out generated/wrapper-api/token-endpoints.json
```

```
npx hardhat generate:wrapper-api-endpoints-config \
  --artifact artifacts/contracts/Account.sol/Account.json \
  --out generated/wrapper-api/account-endpoints.json
```

To start from an ABI file instead, replace `--artifact` with `--abi`, as shown in the following example.

```
npx hardhat generate:wrapper-api-endpoints-config \
  --abi ./abi/Token.json \
  --out generated/wrapper-api/token-endpoints.json
```

Review the generated endpoint configuration before you use it, so that you can choose or refine the methods and paths that are exposed by the wrapper API.

Generate a Wrapper API Postman Collection

Pass one or more endpoint configuration files as a comma-separated value, as shown in the following example.

```
npx hardhat generate:wrapper-api-postman-collection \
  --inputs generated/wrapper-api/token-endpoints.json,generated/wrapper-api/
account-endpoints.json \
  --out generated/wrapper-api/wrapper-api.postman.collection.json
```

Import the resulting collection into Postman and populate its environment variables with the URL and credentials for the deployed wrapper API.

Include an ORDS Swagger/OpenAPI file

The optional `--ords-swagger` argument accepts an ORDS Swagger/OpenAPI JSON file. Its GET operations are added under an ORDS Endpoints folder in the generated Postman collection and are registered as ORDS back end overrides in the generated configuration.

```
npx hardhat generate:wrapper-api-postman-collection \
  --inputs generated/wrapper-api/token-endpoints.json,generated/wrapper-api/
account-endpoints.json \
  --ords-swagger ./api/ords.swagger.json \
  --out generated/wrapper-api/wrapper-api.postman.collection.json
```

Use a valid JSON Swagger/OpenAPI document. The file is not deployed by this command; it is an input used to build the wrapper API and Postman assets.

Define a Combined API

A combined API runs several contract methods in sequence through one POST endpoint. First, generate an endpoint configuration. Then, add a `combinedApis` entry to that configuration, as shown in the following example.

```
{
  "combinedApis": [
    {
      "name": "createAccountWithRole",
      "path": "createAccountWithRole",

```

```

    "httpMethod": "POST",
    "inputs": {
      "required": ["account", "role", "userId"]
    },
    "steps": [
      { "method": "createAccount", "args": ["userId"] },
      { "method": "addRole", "args": ["account", "role"] }
    ],
    "mappings": {
      "createAccount": { "userId": "request.userId" },
      "addRole": {
        "account": "request.account",
        "role": "request.role"
      }
    }
  }
]
}

```

Merge this fragment with the generated configuration; do not replace its generated endpoint content. Use actual ABI method names and argument names from your contract. The request body supplies the named inputs as top-level fields, for example: account, role, and userId. It does not use a payload wrapper.

All steps must belong to the same target configuration. A combined API cannot mix token methods with account or identity methods. Steps run in the order that they are written. Design the sequence and failure handling with that behavior in mind. Then, generate the collection using the edited configuration, as shown in the following example.

```

npx hardhat generate:wrapper-api-postman-collection \
  --inputs generated/wrapper-api/account-endpoints.json \
  --out generated/wrapper-api/account-wrapper-api.postman.collection.json

```

Troubleshooting

Condition	Resolution
Task not found	Check that the project has the wrapper API plug-in installed and configured in the <code>hardhat.config.ts</code> file.
Artifact or ABI file not found	Compile the project when using the <code>--artifact</code> option, or correct the path passed to the <code>--artifact</code> or <code>--abi</code> options.
ORDS endpoints are missing	Verify the <code>--ords-swagger</code> path points to valid JSON and that the source contains supported GET operations.
Combined API validation fails	Ensure that the name, path, steps, required inputs, and mappings are present. Verify that all referenced methods belong to the same token or account/identity configuration.

Generate an RPC Proxy Postman Collection with the Command Line

You can automatically create a Postman collection for a deployed package's token and account/identity ABIs.

The `generate-rpc-proxy-postman` Hardhat task generates a Postman collection for an Oracle Blockchain Platform RPC proxy from account and token contract artifacts. The generated collection contains the proxy requests derived from both ABIs.

Before you can generate the Postman collection, you must install the project dependencies and compile the project so that Hardhat writes the required artifact JSON files. Ensure that the `hardhat.config.ts` file imports the `generate-rpc-proxy-postman` task. Generated projects typically include the following line.

```
import "../tasks/generate-rpc-proxy-postman";
```

There must be one account artifact and one token artifact to run the task. Locate the artifact files under the `artifacts/` folder, and then use their full or project-relative paths in the command.

To generate a Postman collection, run the task with the two artifact paths and an output path, as shown in the following example. Replace the example artifact paths and output filename with those for your project. The output directory is created when necessary.

```
npx hardhat generate-rpc-proxy-postman \
  --account-abi artifacts/contracts/Account.sol/Account.json \
  --token-abi artifacts/contracts/Token.sol/Token.json \
  --out generated/rpc-proxy-postman/token-rpc-proxy.postman.collection.json
```

Do not manually edit the collection. If the ABI changes, regenerate the collection.

To use the generated collection, complete the following steps.

1. In Postman, select **Import** and choose the generated `.postman.collection.json` file.
2. Select or create an environment that supplies the RPC proxy base URL and its authentication values.
3. Review the collection variables and request bodies before sending a transaction. Values such as contract address, signer, and method arguments are environment- or request-specific.

Troubleshooting

Condition	Resolution
Task not found	Confirm that the <code>hardhat.config.ts</code> file imports <code>../tasks/generate-rpc-proxy-postman</code> , then run the command again from the project root.
Artifact file not found	Run the project's compile command and verify the paths under <code>artifacts/</code> folder. Supply the contract artifact JSON, not the source <code>.sol</code> file.

Condition	Resolution
The collection does not include a method	Confirm the method is present in the account or token artifact ABI, then regenerate the collection.
Postman request is unauthorized or cannot reach the proxy	This is an environment configuration issue after collection generation. Verify the selected Postman environment's proxy URL and authentication values.

4

Digital Assets APIs

Oracle Blockchain Platform Enterprise Edition for Besu provides digital assets APIs that you can use to work with smart contracts.

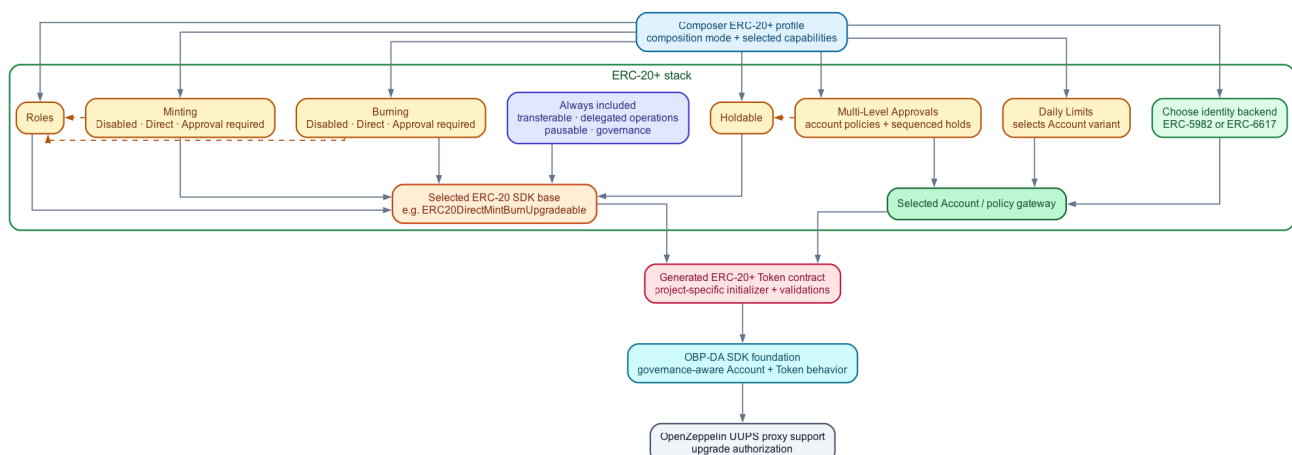
The digital assets APIs are Solidity-based, reusable components for developing, testing, and deploying digital asset smart contracts on Besu. You can use the APIs to implement domain-specific token applications while retaining consistent approaches to token configuration, identity and access controls, upgradeability, and governance.

The core of any Oracle Blockchain Platform Besu application is one or more smart contracts that define the states of a digital asset and the business rules that govern how it is created, transferred, held, redeemed, and managed. Because these rules directly affect the asset life cycle, contracts must be designed, reviewed, and tested before deployment. Each of the two APIs supports a different token standard.

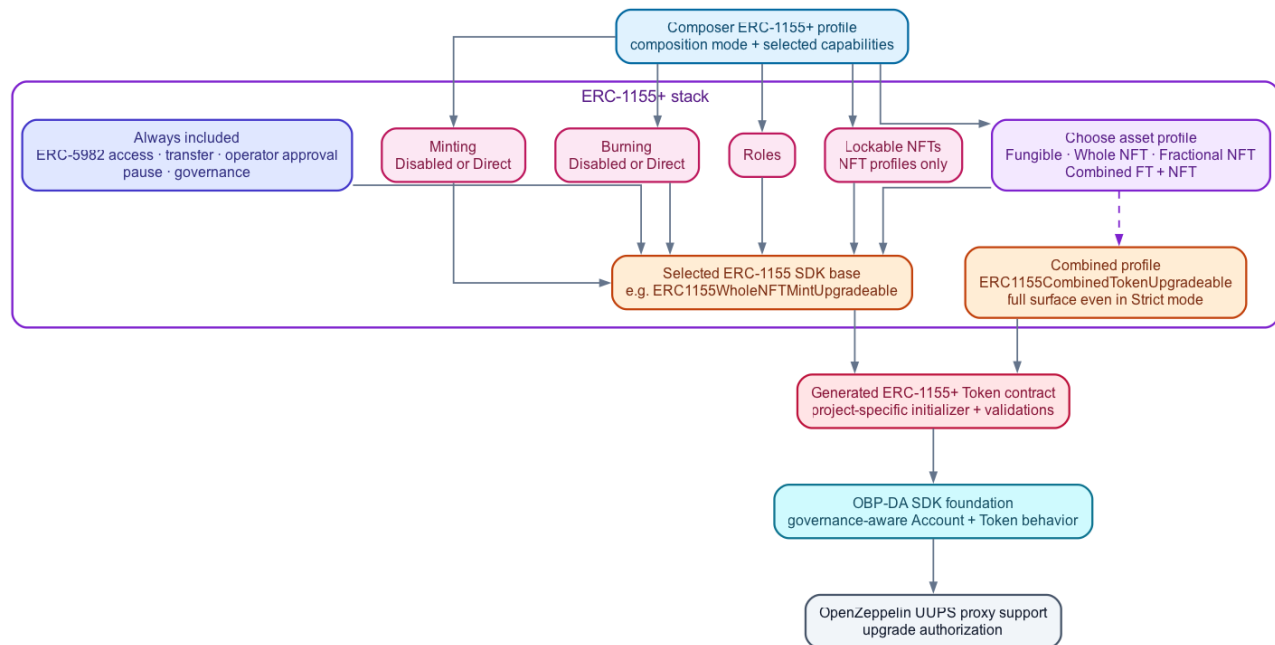
Token Standard	Primary Use	Example Use Cases
ERC-20	Fungible assets where each token is interchangeable with every other token.	Payments, stablecoins, deposit tokens, settlement tokens, and wholesale CBDC assets.
ERC-1155	Fungible assets and non-fungible (NFT) assets. Multiple token types are supported in the same smart contract. Fractional tokens (shares) are supported.	Tokenized assets, collections, and unique digital assets.

The APIs provide configurable base contracts and supporting libraries that you can inherit from or compose into your own Solidity contracts. Depending on the selected token model, applications can incorporate capabilities such as minting and burning, controlled transfers, holds, approvals, identity-aware permissions, and policy enforcement. The APIs also support upgradeable contract patterns and governance-controlled lifecycle operations if these are required by an application.

The following diagram illustrates the capabilities of the extended ERC-20 token standard implementations.



The following diagram illustrates the capabilities of the extended ERC-1155 token standard implementations.



Pluggable Access Control Modules

You can select account and policy modules to align token operations with an application's access-control model. Specifically, the API includes contract and account versions for both the ERC-5982 and the EIP-6617 permission models. These modules connect token contracts to identity and policy gateways, allowing applications to apply role-aware and organization-aware authorization without embedding a single access control implementation in every token contract.

You can choose the account variant and policy capabilities that suit your application, and then use the common account interfaces from the token contracts. This separation keeps token business logic focused on the asset life cycle while allowing access policies to be configured, extended, or exchanged as application requirements evolve.

Contract Composition Model

The Solidity tokenization layer uses a contract composition model. You select an upgradeable token base, add the behavior modules required by the asset lifecycle, and then pair the token with an appropriate account and access-control module. This model supports capabilities such as minting, burning, holds, and approvals without requiring every token to expose every operation. You can use strict composition or mixin / compatibility composition.

Strict composition is appropriate when the supported token operations are known at design time. Developers inherit from a narrow base contract that supports only specific capabilities. Only selected operations are included in the contract ABI. During initialization, the token configuration is validated against the selected capability profile, so that incompatible behaviors or minting and burning modes are rejected.

Mixin / compatibility composition is appropriate when runtime flexibility and a broader, reusable set of token operations are more important. Developers inherit from a fuller base contract, with runtime behavior flags determining which optional policies are active. The full ABI remains

available, while behavior-specific operations enforce the enabled configuration when they are called.

Choose the narrowest composition that meets your product requirements, and then combine it with the required access-control account module and product-specific business logic. Strict composition produces a smaller, more intentional ABI and prevents unsupported operations from being exposed. Mixin / compatibility composition provides a broader foundation for applications whose behavior must be configured at runtime.

Use the APIs as starting points rather than as fixed applications. Typically a team selects the ERC-20 or ERC-1155 foundation that matches the required asset model, configures the behavior and access controls, and then implements specific business logic to support the asset life cycle. This enables a common contract foundation while allowing each application to reflect its own participants, authorization rules, and operational processes.

The APIs are for Solidity developers working directly with the ERC-20 and ERC-1155 contract sources. You can develop and test applications by using Hardhat, and then deploy to an Oracle Blockchain Platform environment by using the deployment approach appropriate for the application.

Governance Flow for Deploying and Upgrading Contracts

Governance controls when a deployed contract becomes usable and which UUPS implementation upgrades are authorized. Composed projects deploy and connect the account and token proxies, and then attach both to a governance contract and governance UUID. The proxies are deliberately inactive until the configured governance flow activates them, as shown in the following life cycle:

1. Deploy account and token UUPS proxies.
2. Set account/token context and governance context. Proxies remain inactive.
3. Submit deployment intent with proxy, implementation, and code-hash components.
4.
 - No-Op: Accepted immediately, components activated.
 - Governed: Collect policy approvals, then components are activated after approval.

Governance Models

The two supported governance models are No-Op and Governed.

No-Op does not remove the governance life cycle from the application. It implements the same governance interface, records the intent, and approves deploy and upgrade checks automatically. Do not use the No-Op model as a production approval mechanism.

The Governed model does not allow the deployer, Blockchain App Builder, or a CLI script to bypass approvers. The required approvers, threshold, deadline, and approval process are defined by the deployed governance and policy configuration.

The following table summarizes the behavior of the two governance models.

Governance Model	Purpose	Deployment Behavior	Upgrade Behavior
No-Op	Local development, demos, and other situations where governance approval is intentionally bypassed.	The <code>proposeDeployIntent</code> function accepts the intent and activates its components immediately.	The upgrade intent is accepted without policy approval; the operator still runs the prepared <code>upgradeToAndCall</code> execution step.

Governance Model	Purpose	Deployment Behavior	Upgrade Behavior
Governed	Environments that require policy-based review and approval.	The intent remains pending until the configured policy receives the required approvals before its deadline; only then are components activated.	The proposed implementation and its code hash must receive the required authorization before the execution step can consume the implementation and upgrade the proxy.

When you build a project with Blockchain App Builder, the following governance building blocks are already supplied:

- The IDAContractGovernance interface and the NoOpGovernance and GovernedGovernance implementations.
- Governance context and activation support on generated token and account contract bases, including `setGovernanceContext`, `getGovernanceContext`, and `activateFromGovernance` methods.
- UUPS upgrade authorization through the API's `UpgradeAuthorizationLib` method, which calls the configured governance contract during the `_authorizeUpgrade` process.
- Generated deployment and upgrade scripts that set governance context, prepare intent data, record manifests, and submit intent requests.

Because these building blocks are already supplied, a standard composed project does not require you to write extra Solidity governance hooks for its generated token and account contracts. You must still choose and deploy or obtain the appropriate governance contract, configure the UUID and policy and provide their addresses to the deployment/upgrade workflow, and undergo the approval process for Governed environments. For an Oracle Blockchain Platform Besu network, a governance contract is deployed as part of instance provisioning.

ERC-20 Account Contract APIs

You can use the following identity and access control methods in ERC-20 account contracts.

Account Management Methods

createAccount

This method creates an account for a specified user. An account must be created for any user who will have tokens at any point. Accounts track balances and on-hold balances. This method can be called only by a Token Admin or by an Org Admin of the specified organization.

Parameters:

- `userId: string` – The user name or email ID of the user. The user ID must not be an empty string.
- `orgId: string` – The membership service provider (MSP) ID of the user in the current organization. The organization ID must not be an empty string.
- `userAddress: string` – The wallet address of the user, which must not be zero.

- `maxDailyAmount: int256` – The maximum amount of tokens that can be used in transactions daily. If the value is -1, there is no limit.
- `maxDailyTransactions: int256` – The maximum number of transactions that can be completed daily. If the value is -1, there is no limit.
- `applicationGroups: string[]` – A list of application groups the user belongs to.

deleteAccount

This method deletes the account of the specified user. This method can be called only by a Token Admin or by an Org Admin of the specified organization.

Parameters:

- `userAddress: string` – The wallet address of the user, which must not be zero.

getAccountStatus

This method gets the current account status of the specified user. This method can be called by the Token Admin or Token Auditor, an Org Admin of the specified organization, or by the token account owner.

Parameters:

- `userAddress: string` – The wallet address of the user, which must not be zero.

Returns:

- A value of the `AccountStatus` enum object, either 0, 1, or 2.

```
enum AccountStatus {
    active,
    suspended,
    deleted
}
```

getAccountByAddress

This method gets the account of the specified user. This method can be called by the Token Admin or Token Auditor, an Org Admin of the specified organization, or by the token account owner.

Parameters:

- `userAddress: string` – The wallet address of the user, which must not be zero.

Return Value Example:

```
{
  "userId": "userA",
  "orgId": "orgA",
  "accountAddress": "0xfbd6c686b912d7722dc86510934589e0aaf3b55a",
  "dailyLimits": {
    "isMaxDailyTxnEnabled": true,
    "isMaxDailyAmtEnabled": true,
    "dailyAmount": 100,
    "dailyTransactions": 5,
    "maxDailyAmount": 100000,
    "maxDailyTransactions": 50,
  }
}
```

```

        "applicationGroups": []
    },
    "status": 0
}

```

getAllAccounts

This method gets all accounts, with paginated output. This method can be called only by the Token Admin or Token Auditor.

Parameters:

- **offset:** uint256 – The offset index that indicates where to start getting account information.
- **limit:** uint256 – The number of items to return.

Returns:

- Return (UserAccount[] memory items, uint256 total, uint256 nextOffset)

Return Value Example:

```

[ {
    "userId": "userA",
    "orgId": "orgA",
    "accountAddress": "0xfbd6c686b912d7722dc86510934589e0aaf3b55a",
    "dailyLimits": {
        "isMaxDailyTxnEnabled": true,
        "isMaxDailyAmtEnabled": true,
        "dailyAmount": 100,
        "dailyTransactions": 5,
        "maxDailyAmount": 100000,
        "maxDailyTransactions": 50,
        "applicationGroups": []
    },
    "status": 0
} ]

```

total: 15

nextoffset: 6

activateAccount

This method activates the account of the specified user. This method can be called only by a Token Admin or by an Org Admin of the specified organization.

Parameters:

- **userAddress:** string – The wallet address of the user, which must not be zero.

suspendAccount

This method activates the account of the specified user. This method can be called only by a Token Admin or by an Org Admin of the specified organization.

Parameters:

- `userAddress`: string – The wallet address of the user, which must not be zero.

setMaxDailyAmount

This method sets the maximum amount of tokens that can be used in transactions daily by the specified user. This method can be called only by a Token Admin or by an Org Admin of the specified organization.

Parameters:

- `value`: int256 – The maximum amount of tokens that can be used in transactions daily.
- `userAddress`: string – The wallet address of the user, which must not be zero.

setMaxDailyTransactionCount

This method sets the maximum number of transactions that can be completed daily by the specified user. This method can be called only by a Token Admin or by an Org Admin of the specified organization.

Parameters:

- `count`: int256 – The maximum number of transactions that can be completed daily.
- `userAddress`: string – The wallet address of the user, which must not be zero.

Admin Management Methods

addTokenAdmin

This method adds a Token Admin. This method can be called only by a Token Admin.

Parameters:

- `adminDetails`: object – An object that specifies the user ID, organization ID, and wallet address, as shown in the following example.

```
{
  "userId": "tokenAdmin1",
  "orgId": "orgA",
  "accountAddress": "0x0fbdc686b912d7722dc86510934589e0aaf3b55a"
}
```

removeTokenAdmin

This method removes a Token Admin. This method can be called only by a Token Admin.

Parameters:

- `userAddress`: string – The wallet address of the user, which must not be zero.

getAllTokenAdmins

This method gets account information for all Token Admin users. This method can be called only by a Token Admin or Token Auditor.

Parameters:

- None

Return Value Example:

```
[{
  "userId": "tokenadmin1",
  "orgId": "orgA",
  "accountAddress": "0x0fbdc686b912d7722dc86510934589e0aaf3b55a"
},
{
  "userId": "tokenadmin2",
  "orgId": "orgB",
  "accountAddress": "0xfe3b557e8fb62b89f4916b721be55ceb828dbd73"
}]
```

isTokenAdmin

This method checks whether a specified user is a Token Admin. This method can be called only by a Token Admin, Token Auditor, Org Admin, or Org Auditor.

Parameters:

- `userAddress`: string – The wallet address of the user, which must not be zero.

Returns:

- A Boolean value (true or false) indicating whether the specified user is a Token Admin.

addOrgAdmin

This method adds an Org Admin. This method can be called only by a Token Admin or Org Admin of the specified organization.

Parameters:

- `adminDetails`: object – An object that specifies the user ID, organization ID, and wallet address, as shown in the following example.

```
{
  "userId": "tokenAdmin1",
  "orgId": "orgA",
  "accountAddress": "0x0fbdc686b912d7722dc86510934589e0aaf3b55a"
}
```

removeOrgAdmin

This method removes an Org Admin. This method can be called only by a Token Admin or Org Admin of the specified organization.

Parameters:

- `userAddress`: string – The wallet address of the user, which must not be zero.

getAllOrgAdmins

This method gets account information for all Org Admin users. This method can be called only by a Token Admin, Token Auditor, Org Admin, or Org Auditor.

Parameters:

- None

Return Value Example:

```
[{
  "userId": "orgadmin1",
  "orgId": "orgA",
  "accountAddress": "0x0fbdc686b912d7722dc86510934589e0aaf3b55a"
},
{
  "userId": "orgadmin2",
  "orgId": "orgB",
  "accountAddress": "0xfe3b557e8fb62b89f4916b721be55ceb828dbd73"
}]
```

isOrgAdmin

This method checks whether a specified user is an Org Admin. This method can be called only by a Token Admin, Token Auditor, Org Admin, or Org Auditor.

Parameters:

- `userAddress`: string – The wallet address of the user, which must not be zero.

Returns:

- A Boolean value (true or false) indicating whether the specified user is an Org Admin.

Auditor Management Methods

addTokenAuditor

This method adds a Token Auditor. This method can be called only by a Token Admin.

Parameters:

- `auditorDetails`: object – An object that specifies the user ID, organization ID, and wallet address, as shown in the following example.

```
{
  "userId": "tokenAuditor1",
  "orgId": "orgA",
  "accountAddress": "0x0fbdc686b912d7722dc86510934589e0aaf3b55a"
}
```

removeTokenAuditor

This method removes a Token Auditor. This method can be called only by a Token Admin.

Parameters:

- `userAddress`: string – The wallet address of the user, which must not be zero.

getAllTokenAuditors

This method gets account information for all Token Auditor users. This method can be called only by a Token Admin or Token Auditor.

Parameters:

- None

Return Value Example:

```
[{
  "userId": "tokenAuditor1",
  "orgId": "orgA",
  "accountAddress": "0x0fbdc686b912d7722dc86510934589e0aaf3b55a"
},
{
  "userId": "tokenAuditor2",
  "orgId": "orgB",
  "accountAddress": "0xfe3b557e8fb62b89f4916b721be55ceb828dbd73"
}]
```

isTokenAuditor

This method checks whether a specified user is a Token Auditor. This method can be called by any user.

Parameters:

- `userAddress`: string – The wallet address of the user, which must not be zero.

Returns:

- A Boolean value (true or false) indicating whether the specified user is a Token Auditor.

addOrgAuditor

This method adds an Org Auditor. This method can be called only by a Token Admin or Org Admin of the specified organization.

Parameters:

- `adminDetails`: object – An object that specifies the user ID, organization ID, and wallet address, as shown in the following example.

```
{
  "userId": "tokenAuditor1",
  "orgId": "orgA",
  "accountAddress": "0x0fbdc686b912d7722dc86510934589e0aaf3b55a"
}
```

removeOrgAuditor

This method removes an Org Auditor. This method can be called only by a Token Admin or Org Admin of the specified organization.

Parameters:

- `userAddress`: string – The wallet address of the user, which must not be zero.

getAllOrgAuditors

This method gets account information for all Org Auditor users. This method can be called only by a Token Admin, Token Auditor, Org Admin, or Org Auditor.

Parameters:

- None

Return Value Example:

```
[{
  "userId": "orgAuditor1",
  "orgId": "orgA",
  "accountAddress": "0xf0bdc686b912d7722dc86510934589e0aaf3b55a"
},
{
  "userId": "orgAuditor2",
  "orgId": "orgB",
  "accountAddress": "0xfe3b557e8fb62b89f4916b721be55ceb828dbd73"
}]
```

isOrgAuditor

This method checks whether a specified user is an Org Auditor. This method can be called by any user.

Parameters:

- `userAddress`: string – The wallet address of the user, which must not be zero.

Returns:

- A Boolean value (true or false) indicating whether the specified user is an Org Auditor.

Role Management Methods

addRole

This method adds a role to a specified user. This method can be called only by a Token Admin or Org Admin of the specified organization.

Parameters:

- `userAddress`: string – The wallet address of the user, which must not be zero.
- `role`: string – The name of the role to add for the specified user, such as minter, burner, or notary.

removeRole

This method removes a role from the specified user. This method can be called only by a Token Admin or Org Admin of the specified organization.

Parameters:

- `userAddress`: string – The wallet address of the user, which must not be zero.
- `role`: string – The name of the role to remove for the specified user, such as minter, burner, or notary.

isInRole

This method checks whether a user has a specified role. This method can be called by any user.

Parameters:

- `role`: string – The name of the role to check for the specified user, such as minter, burner, or notary.

- `userAddress: string` – The wallet address of the user, which must not be zero.

Returns:

- A Boolean value (true or false) indicating whether the user has the specified role.

hasRole

This method checks whether a user has a specified role. This method can be called by any user.

Parameters:

- `account: string` – The wallet address of the user, which must not be zero.
- `role: bytes32` – The name of the role to check for the specified user, such as `minter`, `burner`, or `notary`.

Returns:

- A Boolean value (true or false) indicating whether the user has the specified role.

Context Management Methods

setContext

This method sets the token contract address for the account contract. This method can be called only by a `Token Admin`.

Parameters:

- `tokenAddress: address` – The contract address of the deployed token, which must not be zero.

getContext

This method gets the token contract address for the account contract. This method can be called by any user.

Parameters:

- `None`

Returns

- The token contract address

setGovernanceContext

This method sets the governance contract to use to manage the governance life cycle for the account contract. This method can be called only by a `Token Admin`.

Parameters:

- `uuid: string` – The unique ID set by the digital assets `Token Admin` before contract deployment.
- `governance: address` – The contract address of the deployed governance contract, which must not be zero.

getGovernanceContext

This method gets the governance context details. This method can be called by any user.

Parameters:

- None

Returns

- A `governanceContext` object.

Return Value Example:

```
{
  "governanceUUID": "abc-def",
  "governance": "0x00000000",
  "deploymentActive": "true"
}
```

isDeploymentActive

This method checks whether the deployment is active. This method can be called by any user.

Parameters:

- None

Returns:

- A Boolean value (true or false) indicating whether the deployment is active.

ERC-20 Token Contract APIs

You can use the following methods related to token functionality in ERC-20 token contracts.

Token Configuration Methods

`__ERC20Token_init`

This method is called when a token contract is deployed. This method can be called only by a Token Admin.

Parameters:

- `name`: string – The name of the token.
- `symbol`: string – The symbol of the token.

`initializeERC20Token`

This method initializes an ERC-20 token. This method can be called only by a Token Admin.

Parameters:

- `token`: `ERC20Token` – The structure defining the ERC-20 token, as shown in the following example.

```
{
  "tokenId": "WCBDC-100",
  "tokenName": "wholesale cbdc",
  "tokenDesc": "this is wcbdc contract",
  "tokenStandard": "ttf+",
  "tokenType": "fungible",
}
```

```
        "tokenUnit": "fractional",
        "behaviors": ["mintable", "burnable", "transferable", "roles",
"holdable", "pausable"],
        "decimals": 2,
        "mintable": { "maxMintQuantity": 1000000, "mintApprovalRequired":
false },
        "burnable": { "burnApprovalRequired": false }
    }
```

getToken

This method gets an ERC-20 token. This method can be called only by a Token Admin, Token Auditor, Org Admin, or Org Auditor.

Parameters:

- None

Return Value Example:

```
{
  "tokenId": "WCBDC-100",
  "tokenName": "wholesale cbdc",
  "tokenDesc": "this is wcbdc contract",
  "tokenStandard": "ttf+",
  "tokenType": "fungible",
  "tokenUnit": "fractional",
  "behaviors": ["mintable", "burnable", "transferable", "roles",
"holdable", "pausable"],
  "decimals": 2,
  "mintable": { "maxMintQuantity": 1000000, "mintApprovalRequired": false },
  "burnable": { "burnApprovalRequired": false }
}
```

decimals

This method returns the number of decimals places that were configured for a token. This method can be called only by a Token Admin, Token Auditor, Org Admin, or Org Auditor.

Parameters:

- None

Returns:

- A uint8 value indicating the number of decimal places.

cap

This method returns the cap value (the maximum total supply) for an ERC-20 token. This method can be called only by a Token Admin, Token Auditor, Org Admin, or Org Auditor.

Parameters:

- None

Returns:

- A uint256 value indicating the cap value.

balanceOf

This method returns the current token balance for a specified user. This method can be called only by a Token Admin or Token Auditor, or the Org Admin or Org Auditor of the specified organization.

Parameters:

- `userAddress: string` – The wallet address of the user, which must not be zero.

Returns:

- A `uint256` value indicating the current balance.

Token Behavior Methods - Mintable Behavior**mint**

This method mints ERC-20 tokens. This method can be called by any user with the minter role.

Parameters:

- `to: string` – The wallet address of the user the tokens are being minted for, which must not be zero.
- `value: uint256` – The quantity of tokens to mint.

requestMint

This method can be called by any user with the minter role to send a request to the notary to create a specified amount of tokens.

Parameters:

- `notary: string` – The wallet address of the notary user, which must not be zero.
- `amount: uint256` – The quantity of tokens to mint.
- `expiration: uint256` – The expiration time of the request in epoch format.
- `opId: string` – The operation ID of the request.
- `info_details: JSON` – An object specifying the category (category) and description (description) of the request.

You specify the `info_details` parameter in a different format if you are using Visual Studio Code versus the CLI or a Postman collection.

Visual Studio Code: `{ "category": "category value", "description": "description value" }`

CLI / Postman: `{"category\":\"category value\",\"description\":\"description value\"}"`

approveMint

This method can be called by an approver with the notary role to approve a mint request for ERC-20 tokens.

Parameters:

- `opId: string` – The operation ID of the mint request.

rejectMint

This method can be called by an approver with the notary role to reject a mint request for ERC-20 tokens.

Parameters:

- `opId: string` – The operation ID of the mint request.

Token Behavior Methods - Burnable Behavior

burn

This method burns ERC-20 tokens. This method can be called by any user with the burner role.

Parameters:

- `account: string` – The wallet address of the user the tokens are being burned for, which must not be zero.
- `value: uint256` – The quantity of tokens to burn.

burnFrom

This method can be called by any user with a delegated allowance.

Parameters:

- `account: string` – The wallet address of the account to burn tokens from, which must not be zero.
- `value: uint256` – The quantity of tokens to burn.

requestBurn

This method can be called by any user with a burner role to approve a burn request for ERC-20 tokens.

Parameters:

- `notary: string` – The wallet address of the notary user, which must not be zero.
- `amount: uint256` – The quantity of tokens to burn.
- `expiration: uint256` – The expiration time of the request in epoch format.
- `opId: string` – The operation ID of the request.
- `info_details: JSON` – An object specifying the category (category) and description (description) of the request.

You specify the `info_details` parameter in a different format if you are using Visual Studio Code versus the CLI or a Postman collection.

Visual Studio Code: { "category": "category value", "description": "description value" }

CLI / Postman: {"category\\":\\"category value\\",\\"description\\":\\"description value\\"}"

approveBurn

This method can be called by an approver with the notary role to approve a burn request for ERC-20 tokens.

Parameters:

- `opId: string` – The operation ID of the burn request.

rejectBurn

This method can be called by an approver with the notary role to reject a burn request for ERC-20 tokens.

Parameters:

- `opId: string` – The operation ID of the burn request.

Token Behavior Methods - Transferable Behavior**transfer**

This method can be called by any user with tokens to transfer them to another user.

Parameters:

- `to: string` – The wallet address of the receiver, which must not be zero.
- `value: uint256` – The quantity of tokens to transfer.

batchTransfer

This method can be called by any user with tokens to transfer them in batches to other users.

Parameters:

- `toList: string[]` – A list of wallet addresses of the receivers.
- `amounts: uint256[]` – A list of the quantity of tokens to transfer.

Token Behavior Methods - Delegable Behavior**allowance**

This method can be called by any user who owns tokens to delegate another user to spend them.

Parameters:

- `owner: string` – The wallet address of the token owner, which must not be zero.
- `spender: string` – The wallet address of the token spender, which must not be zero.

Returns:

- A `uint256` value of the amount of tokens delegated to be spent.

approve

This method can be called by any user who owns tokens to set the amount that a delegated spender can spend.

Parameters:

- `spender: string` – The wallet address of the token spender, which must not be zero.
- `value: uint256` – The amount of tokens that the spender is allowed to spend.

transferFrom

This method can be called by a user with a delegated allowance to transfer tokens to another user.

Parameters:

- `from: string` – The wallet address of the sender, which must not be zero.

- **to:** string – The wallet address of the receiver, which must not be zero.
- **value:** uint256 – The quantity of tokens to transfer.

Token Behavior Methods - Pausable Behavior

paused

This method checks whether a contract is paused. This method can be called only by a Token Admin, Token Auditor, Org Admin, or Org Auditor.

Parameters:

- None

pause

This method pauses a contract. This method can be called only by a Token Admin.

Parameters:

- None

unpause

This method starts a paused contract. This method can be called only by a Token Admin.

Parameters:

- None

Token Behavior Methods - Holdable Behavior

hold

This method can be called by any user with tokens to request a token transfer.

Parameters:

- **to:** string – The wallet address of the receiver, which must not be zero.
- **notary:** string – The wallet address of the notary user, which must not be zero.
- **amount:** uint256 – The quantity of tokens to transfer.
- **expiration:** uint256 – The expiration time of the request in epoch format.
- **opId:** string – The operation ID of the request.
- **holdType:** string – The type of hold. For example, transfer.
- **info_details:** JSON – An object specifying the category (category) and description (description) of the request.

You specify the `info_details` parameter in a different format if you are using Visual Studio Code versus the CLI or a Postman collection.

Visual Studio Code: { "category": "category value", "description": "description value" }

CLI / Postman: {"category\\":\\"category value\\",\\"description\\":\\"description value\\"}"

holdFrom

This method can be called by a delegated user with an allowance to hold tokens from another user's wallet address. This method requires the delegable behavior.

Parameters:

- `fromAccount`: string – The wallet address of the user whose tokens will be held, which must not be zero.
- `toAccount`: string – The wallet address of the receiver, which must not be zero.
- `notary`: string – The wallet address of the notary user, which must not be zero.
- `amount`: uint256 – The quantity of tokens to transfer.
- `expiration`: uint256 – The expiration time of the request in epoch format.
- `opId`: string – The operation ID of the request.
- `holdType`: string – The type of hold. For example, transfer.
- `info_details`: JSON – An object specifying the category (category) and description (description) of the request.

You specify the `info_details` parameter in a different format if you are using Visual Studio Code versus the CLI or a Postman collection.

Visual Studio Code: { "category": "category value", "description": "description value" }

CLI / Postman: {"category\":"category value\","description\":"description value\"}"

executeHold

This method can be called by an approver with the notary role to approve a hold request.

Parameters:

- `amount`: uint256 – The quantity of tokens to approve for transfer.
- `opId`: string – The operation ID of the hold request.

releaseHold

This method can be called by an approver with the notary role to reject a hold request.

Parameters:

- `opId`: string – The operation ID of the hold request.

updateNotary

This method can be called by an approver with the notary role to update the notary for a specified request.

Parameters:

- `opId`: string – The operation ID of the hold request.
- `newNotary`: string – The wallet address of the new notary user, which must not be zero.

getOnHoldBalanceWithOperationId

This method gets the hold balance for a specified operation ID. This method can be called only by a Token Admin or Token Auditor, Org Admin or Org Auditor of the specified organization, or by a transaction participant (sender, recipient, notary).

Parameters:

- `opId`: string – The operation ID of the hold request.

Returns:

- A uint256 value of the hold balance.

getOnHoldDetailsWithOperationId

This method gets the hold details for a specified operation ID. This method can be called only by a Token Admin or Token Auditor, Org Admin or Org Auditor of the specified organization, or by a transaction participant (sender, recipient, notary).

Parameters:

- opId: string – The operation ID of the hold request.

Returns:

- A hold object, with the following information.
 - to: address – The wallet address of the receiver.
 - notary: address – The wallet address of the notary.
 - amount: uint256 – The quantity of tokens to be transferred.
 - expiration: uint256 – The expiration time for the request, in epoch format.
 - opId: string – The operation ID of the request.
 - holdType: string – The hold type; for example, transfer.
 - info: infoDetails – An object specifying the category (category) and description (description) of the request.

getAccountOnHoldBalance

This method gets the hold balance for a specified account. This method can be called only by a Token Admin or Token Auditor, Org Admin or Org Auditor of the specified organization, or by a transaction participant (sender, recipient, notary).

Parameters:

- account: string – The wallet address of the account to check, which must not be zero.

Returns:

- A uint256 value of the hold balance.

Token Behavior Methods - Multi-Level Approval Behavior

createOrupdateAccountPolicy

This method can be called by a Token Admin or Org Admin to create or update a policy for a specified account.

Parameters:

- accountPolicyId – The system generates this ID from the specified orgId and userId fields. You do not provide this field manually.
- orgId – The membership service provider (MSP) ID of the user to create the policy for.
- userId – The user name or email ID of the user to create the policy for.
- kycCompliance – A string value (true or false) that indicates whether the account satisfies KYC (Know Your Customer) requirements.

- **amlCompliance** – A string value (true or false) that indicates whether the account satisfies AML (Anti-Money Laundering) requirements.
- **riskScore** – The risk score associated with the account, which is used for compliance evaluation.
- **restrictionFlag** – A string value (true or false) that indicates whether the account is subject to restricted transfers. If set to true, direct transfers follow the transfer restriction bucket limits and hold transfers follow the lowest approval-policy threshold.

getAccountPolicyById

This method gets account policy details for a specified policy ID. This method can be called by a Token Admin or Token Auditor, or Org Admin or Org Auditor of the organization whose account policy will be fetched.

Parameters:

- **accountPolicyId**: string – The unique account policy ID.

Returns:

- **accountPolicyId** – The system generates this ID from the specified orgId and userId fields. You do not provide this field manually.
- **orgId** – The membership service provider (MSP) ID of the user.
- **userId** – The user name or email ID of the user.
- **kycCompliance** – A string value (true or false) that indicates whether the account satisfies KYC (Know Your Customer) requirements.
- **amlCompliance** – A string value (true or false) that indicates whether the account satisfies AML (Anti-Money Laundering) requirements.
- **riskScore** – The risk score associated with the account, which is used for compliance evaluation.
- **restrictionFlag** – A string value (true or false) that indicates whether the account is subject to restricted transfers. If set to true, direct transfers follow the transfer restriction bucket limits and hold transfers follow the lowest approval-policy threshold.

getAccountPolicyById

This method gets account policy details for a specified user. This method can be called by any user.

Parameters:

- **who**: address – The wallet address of the user.

Returns:

- The accountPolicyId string.

deleteAccountPolicyById

This method deletes a policy for a specified policy ID. This method can be called by a Token Admin, or Org Admin of the organization that includes the user whose account policy will be deleted.

Parameters:

- **accountPolicyId** – The unique account policy ID.

Returns:

- The accountPolicyId string.

createOrupdateApprovalPolicy

This method can be called by a Token Admin.

Parameters:

- approvalPolicyId – The system generates this ID. You do not provide this field manually.
- transactionLowerLimit – The minimum transaction amount the approval policy applies to.
- transactionUpperLimit – The maximum transaction amount the approval policy applies to.
- numberOfApprovalsRequired – The total number of approvals needed before the transaction can be completed.
- approverDetails – A list of approvers along with their assigned approval sequence, which defines the mandatory order for approvals.
- version – The approval policy version. Use 0 when creating a fresh policy.
- status – Active or inactive.

getApprovalPolicyById

This method gets approval policy details for a specified policy ID. This method can be called by a Token Admin or Token Auditor.

Parameters:

- approvalPolicyId: string – The unique approval policy ID.

Returns:

- approvalPolicyId – The system generates this ID.
- transactionLowerLimit – The minimum transaction amount the approval policy applies to.
- transactionUpperLimit – The maximum transaction amount the approval policy applies to.
- numberOfApprovalsRequired – The total number of approvals needed before the transaction can be completed.
- approverDetails – A list of approvers along with their assigned approval sequence, which defines the mandatory order for approvals.
- version – The approval policy version.
- status – Active or inactive.

deleteApprovalPolicy

This method deletes a policy for a specified policy ID. This method can be called by a Token Admin.

Parameters:

- approvalPolicyId: string – The unique approval policy ID.

getApprovalTransactionsById

This method gets the on-hold balance for a specified account. This method can be called by a Token Admin, Token Auditor, Org Admin, or Org Auditor.

Parameters:

- opId: string – The operation ID for the hold request.

Returns:

- An approvalTransactions object.

Return Value Example:

```
{
  "approvalTransactionId": "",
  "approvalPolicyId": "",
  "approvalPolicyVersion": "",
  "fromAccount": "",
  "toAccount": "",
  "totalApprovals": "",
  "numeraOfApprovalsRequired": "",
  "status": "",
  "approverDetails": ""
}
```

getApprovalStatusByOperationId

This method gets approval details for a specified hold request. This method can be called by a Token Admin, Token Auditor, Org Admin, or Org Auditor.

Parameters:

- opId: string – The operation ID for the hold request.

Return Value Example:

```
{
  "approvalWorkflowExists": "",
  "activePolicyFound": "",
  "receivedApprovals": "",
  "requiredApprovals": "",
  "approvalRequirementMet": "",
  "nextApprovalSequence": "",
  "nextApprover": "",
  "workflowStatus": "",
  "approvalPolicyId": "",
  "approvalPolicyVersion": ""
}
```

approveTransaction

An approver can use this method to approve a hold transaction.

Parameters:

- opId: string – The operation ID for the hold request.

- `amount: uint256` – The amount to approve.

rejectTransaction

A notary can use this method to reject a hold transaction.

Parameters:

- `opId: string` – The operation ID for the hold request.

getTransferRestriction

This method returns the current transfer restriction configuration, including the lower and upper transaction limits. This method can be called by a Token Admin or Token Auditor.

Parameters:

- `account: address` – The wallet address of the user to check the on-hold balance for. The address must not be zero.

Returns:

- A `transferRestriction` object.

Return Value Example:

```
transactionLowerLimit: 0transactionUpperLimit: 100
```

setOrupdateTransferRestriction

This method updates the transfer restriction configuration by setting new lower and upper transaction limits. This method can be called by a Token Admin.

Parameters:

- `transactionLowerLimit: uint256` – The minimum allowed transaction amount.
- `transactionUpperLimit: uint256` – The maximum allowed transaction amount.

Context Management Methods**setTokenContext**

This method sets the token contract address for the account contract. This method can be called only by a Token Admin.

Parameters:

- `tokenAddress: address` – The contract address of the deployed token, which must not be zero.

getTokenContext

This method gets the token contract address for the account contract. This method can be called by any user.

Parameters:

- None

Returns

- The token contract address

setGovernanceContext

This method sets the governance contract to use to manage the governance life cycle for the account contract. This method can be called only by a Token Admin.

Parameters:

- `uuid: string` – The unique ID set by the digital assets Token Admin before contract deployment.
- `governance: address` – The contract address of the deployed governance contract, which must not be zero.

getGovernanceContext

This method gets the governance context details. This method can be called by any user.

Parameters:

- None

Returns

- A `governanceContext` object.

Return Value Example:

```
{
  "governanceUUID": "abc-def",
  "governance": "0x00000000",
  "deploymentActive": "true"
}
```

isDeploymentActive

This method checks whether the deployment is active. This method can be called by any user.

Parameters:

- None

Returns:

- A Boolean value (true or false) indicating whether the deployment is active.

ERC-1155 Account Contract APIs

You can use the following identity and access control methods in ERC-1155 account contracts.

Account Management Methods

createAccount

This method creates an account for a specified user. An account must be created for any user who will have tokens at any point. Accounts track balances and on-hold balances. This method can be called only by a Token Admin or by an Org Admin of the specified organization.

Parameters:

- `userId: string` – The user name or email ID of the user. The user ID must not be an empty string.

- `orgId`: string – The membership service provider (MSP) ID of the user in the current organization. The organization ID must not be an empty string.
- `userAddress`: string – The wallet address of the user, which must not be zero.

deleteAccount

This method deletes the account of the specified user. This method can be called only by a Token Admin.

Parameters:

- `userAddress`: string – The wallet address of the user, which must not be zero.

getAccountStatus

This method gets the current account status of the specified user. This method can be called by the Token Admin or by the token account owner.

Parameters:

- `userAddress`: string – The wallet address of the user, which must not be zero.

Returns:

- A value of the `AccountStatus` enum object, either 0, 1, or 2.

```
enum AccountStatus {  
    active,  
    suspended,  
    deleted  
}
```

getAccountByAddress

This method gets the account of the specified user. This method can be called by the Token Admin or by the token account owner.

Parameters:

- `userAddress`: string – The wallet address of the user, which must not be zero.

Return Value Example:

```
{  
  "userId": "userA",  
  "orgId": "orgA",  
  "accountAddress": "0x0fbdc686b912d7722dc86510934589e0aaf3b55a",  
  "status": 0  
}
```

getAllAccounts

This method gets all accounts, with paginated output. This method can be called only by the Token Admin.

Parameters:

- `offset`: uint256 – The offset index that indicates where to start getting account information.

- `limit: uint256` – The number of items to return.

Returns:

- Return (`UserAccount[]` memory items, `uint256` total, `uint256` nextOffset)

Return Value Example:

```
[{
  "userId": "userA",
  "orgId": "orgA",
  "accountAddress": "0xfbd6686b912d7722dc86510934589e0aaf3b55a",
  "status": 0
}]
```

total: 15

nextoffset: 6

activateAccount

This method activates the account of the specified user. This method can be called only by a Token Admin.

Parameters:

- `userAddress: string` – The wallet address of the user, which must not be zero.

suspendAccount

This method activates the account of the specified user. This method can be called only by a Token Admin.

Parameters:

- `userAddress: string` – The wallet address of the user, which must not be zero.

Admin Management Methods

addTokenAdmin

This method adds a Token Admin. This method can be called only by a Token Admin.

Parameters:

- `adminDetails: object` – An object that specifies the user ID, organization ID, and wallet address, as shown in the following example.

```
{
  "userId": "tokenAdmin1",
  "orgId": "orgA",
  "accountAddress": "0xfbd6686b912d7722dc86510934589e0aaf3b55a"
}
```

removeTokenAdmin

This method removes a Token Admin. This method can be called only by a Token Admin.

Parameters:

- `userAddress: string` – The wallet address of the user, which must not be zero.

getAllTokenAdmins

This method gets account information for all Token Admin users. This method can be called only by a Token Admin.

Parameters:

- None

Return Value Example:

```
[{
  "userId": "tokenadmin1",
  "orgId": "orgA",
  "accountAddress": "0xfbd6c686b912d7722dc86510934589e0aaf3b55a"
},
{
  "userId": "tokenadmin2",
  "orgId": "orgB",
  "accountAddress": "0xfe3b557e8fb62b89f4916b721be55ceb828dbd73"
}]
```

isTokenAdmin

This method checks whether a specified user is a Token Admin. This method can be called only by a Token Admin.

Parameters:

- `userAddress: string` – The wallet address of the user, which must not be zero.

Returns:

- A Boolean value (true or false) indicating whether the specified user is a Token Admin.

Role Management Methods**addRole**

This method adds a role to a specified user. This method can be called only by a Token Admin.

Parameters:

- `userAddress: string` – The wallet address of the user, which must not be zero.
- `role: string` – The name of the role to add for the specified user, such as `minter`, `burner`, or `notary`.
- `scopeId: uint256` – The scope ID, which is either an NFT class ID or a fungible token ID.

removeRole

This method removes a role from the specified user. This method can be called only by a Token Admin.

Parameters:

- `userAddress: string` – The wallet address of the user, which must not be zero.

- `role: string` – The name of the role to remove for the specified user, such as `minter`, `burner`, or `notary`.
- `scopeId: uint256` – The scope ID, which is either an NFT class ID or a fungible token ID.

isInRole

This method checks whether a user has a specified role. This method can be called only by a `Token Admin`.

Parameters:

- `role: string` – The name of the role to check for the specified user, such as `minter`, `burner`, or `notary`.
- `userAddress: string` – The wallet address of the user, which must not be zero.
- `scopeId: uint256` – The scope ID, which is either an NFT class ID or a fungible token ID.

Returns:

- A Boolean value (`true` or `false`) indicating whether the user has the specified role.

isInRole

This method checks whether a user has a specified role. This method can be called only by a `Token Admin`.

Parameters:

- `role: bytes32` – The Keccak-256 hash of the name of the role to check for the specified user, such as `minter`, `burner`, or `notary`.
- `userAddress: string` – The wallet address of the user, which must not be zero.
- `scopeId: uint256` – The scope ID, which is either an NFT class ID or a fungible token ID.

Returns:

- A Boolean value (`true` or `false`) indicating whether the user has the specified role.

addTokenSysRole

This method adds the `TOKEN_SYS_VAULT_ROLE` role to a specified user. This method can be called only by a `Token Admin`.

Parameters:

- `userAddress: string` – The wallet address of the user, which must not be zero.
- `role: string` – The name of the role to add. The only supported value is `TOKEN_SYS_VAULT_ROLE`.

Returns:

- A Boolean value (`true` or `false`).

removeTokenSysRole

This method removes the `TOKEN_SYS_VAULT_ROLE` role from a specified user. This method can be called only by a `Token Admin`.

Parameters:

- `userAddress: string` – The wallet address of the user, which must not be zero.

- `role: string` – The name of the role to remove. The only supported value is `TOKEN_SYS_VAULT_ROLE`.

Returns:

- A Boolean value (true or false).

transferTokenSysRole

This method transfers the `TOKEN_SYS_VAULT_ROLE` role from a specified user to a recipient. This method can be called only by a Token Admin.

Parameters:

- `fromAddress: string` – The wallet address of the user who currently has the `TOKEN_SYS_VAULT_ROLE` role.
- `toAddress: string` – The wallet address of the user who will receive the `TOKEN_SYS_VAULT_ROLE` role.
- `role: string` – The name of the role to transfer. The only supported value is `TOKEN_SYS_VAULT_ROLE`.

Returns:

- A Boolean value (true or false).

Context Management Methods

setContext

This method sets the token contract address for the account contract. This method can be called only by a Token Admin.

Parameters:

- `tokenAddress: address` – The contract address of the deployed token, which must not be zero.

getContext

This method gets the token contract address for the account contract. This method can be called by any user.

Parameters:

- None

Returns

- The token contract address

setGovernanceContext

This method sets the governance contract to use to manage the governance life cycle for the account contract. This method can be called only by a Token Admin.

Parameters:

- `uuid: string` – The unique ID set by the digital assets Token Admin before contract deployment.
- `governance: address` – The contract address of the deployed governance contract, which must not be zero.

getGovernanceContext

This method gets the governance context details. This method can be called by any user.

Parameters:

- None

Returns

- A governanceContext object.

Return Value Example:

```
{
  "governanceUUID": "abc-def",
  "governance": "0x00000000",
  "deploymentActive": "true"
}
```

isDeploymentActive

This method checks whether the deployment is active. This method can be called by any user.

Parameters:

- None

Returns:

- A Boolean value (true or false) indicating whether the deployment is active.

ERC-1155 Token Contract APIs

You can use the following methods related to token functionality in ERC-1155 token contracts.

Token Configuration Methods**__ERC1155Token_init**

This method is called when a token contract is deployed. This method can be called only by a Token Admin.

Parameters:

- uri: string – The URI for all token types.

Token Setup Methods**saveNFTClass**

This method saves NFT class information on the ledger. This method can be called only by a Token Admin.

Parameters:

- `classInfo`: `NFTClassInfo` – The account contract address, which varies depending on whether the NFT is whole or fractional. Whole NFT:

```
{
  "nftClassId": (BigInt(1)),
  "tokenName": "ArtCollection2",
  "tokenDesc": "this is art collection contract",
  "tokenStandard": "erc1155",
  "tokenUnit": 0,
  "behaviors": ["mintable", "burnable", "transferable", "roles",
"indivisible"],
  "divisible": { "decimals": 0 },
  "mintable": { "maxMintQuantity": 2 }
}
```

Fractional NFT:

```
"nftClassId": (BigInt(0)),
tokenName": "ArtCollection",
"tokenDesc": "this is art collection contract",
"tokenStandard": "erc1155",
"tokenUnit": 1,
"behaviors": ["mintable", "burnable", "transferable", "roles",
"divisible"],
"divisible": { "decimals": 0 },
"mintable": { "maxMintQuantity": 1 }
```

createNonFungibleToken

This method mints NFTs based on the saved NFT class information. This method can be called by a user with the `minter` role or by the `Token Admin`.

Parameters:

- `token`: `uint256` – The ID of the NFT, which contains NFT class information. The top 128 bits of the `tokenId` parameter represent the NFT class ID, while the bottom 128 bits represent the unique index of the NFT.
- `quantity`: `uint256` – The amount of the NFT to mint (create).

createFungibleToken

This method is called when the token contract is deployed. This method can be called only by the `Token Admin`.

Parameters:

- `tokenId`: `ERC1155Token` – The fungible token definition to record in the ledger, as shown in the following example.

```
"tokenId": BigInt(0),
"tokenName": "wcbdc",
"tokenDesc": "this is wcbdc token",
"tokenStandard": "erc1155",
"tokenType": 1,
```

```

    "tokenUnit": 1,
    "behaviors": ["mintable", "burnable", "transferable", "roles",
"divisible"],
    "divisible": { "decimals": 2 },
    "quantity": 1,
    "mintable": { "maxMintQuantity": 1 }

```

getTokenById

This method gets the details for a specified token. This method can be called only by a Token Admin or by a token owner.

Parameters:

- token: ERC1155Token – The token definition, as shown in the following example:

```

    "tokenId": BigInt(0),
    "tokenName": "wcbdc",
    "tokenDesc": "this is wcbdc token",
    "tokenStandard": "erc1155",
    "tokenType": 1,
    "tokenUnit": 1,
    "behaviors": ["mintable", "burnable", "transferable", "roles",
"divisible"],
    "divisible": { "decimals": 2 },
    "quantity": 1,
    "mintable": { "maxMintQuantity": 1 }

```

- metaInfo: bytes – For fungible tokens, the metaInfo parameter is empty. For whole NFTs, the metaInfo parameter contains encoded data for the following structure:

```

struct ERC1155WholeNFT {
    bool isBurned;
    bool isLocked;
    uint256 creationDate;
    uint256 quantity;
    address createdBy;
    address owner;
}

```

For fractional NFTs, the metaInfo parameter contains encoded data for the following structure:

```

struct ERC1155FractionalNFT {
    bool isBurned;
    bool isLocked;
    uint256 creationDate;
    uint256 quantity;
    address createdBy;
    Owners[] owners;
}

```

getTokenDecimals

This method returns the number of decimals places that were configured for a token. This method can be called only by a Token Admin.

Parameters:

- `id: uint256` – The token ID.

Returns:

- A `uint8` value indicating the number of decimal places.

tokenIdOf

This method returns the token ID. This method can be called only by a Token Admin.

Parameters:

- `classId: uint256` – The NFT class ID.
- `serialId: uint256` – The serial ID in the NFT class.

Returns:

- A `uint256` value of the token ID.

balanceOf

This method returns the current token balance for a specified user. This method can be called only by a Token Admin or by the token owner.

Parameters:

- `account: string` – The account address of the user.
- `id: uint256` – The token ID.

Returns:

- A `uint256` value indicating the current balance.

balanceOfBatch

This method returns the current token balances for a batch of specified users. This method can be called only by a Token Admin or by the token owner.

Parameters:

- `accounts: string[]` – A list of account addresses of users.
- `ids: uint256[]` – A list of token IDs.

Returns:

- A `uint256[]` list of values indicating the current balances.

exists

This method checks whether a specified token exists. This method can be called only by a Token Admin.

Parameters:

- `id: uint256` – The token ID.

Returns:

- A Boolean value (true or false).

totalSupply

This method returns the total supply of tokens in a contract. This method can be called only by a Token Admin.

Parameters:

- `id: uint256` (optional) – A token ID. If you do not specify a token ID, the total supply of all tokens in the contract is returned

Returns:

- A `uint256` value indicating the total supply.

Token Behavior Methods - Mintable Behavior**mintBatch**

This method mints ERC-1155 tokens in batches. The tokens must already be initialized. This method can be called by any user with the minter role.

Parameters:

- `tokenIds: uint256[]` – A list of token IDs to mint.
- `quantity: uint256[]` – A corresponding list of token quantities to mint.
- `data: bytes` – Additional data in byte format.

Token Behavior Methods - Burnable Behavior**burnBatch**

This method burns ERC-1155 tokens in batches. The tokens must already be initialized. This method can be called by any user with the burner role.

Parameters:

- `tokenIds: uint256[]` – A list of token IDs to burn.
- `quantity: uint256[]` – A corresponding list of token quantities to burn.
- `data: bytes` – Additional data in byte format.

burnNFT

This method burns NFTs. The token must already be initialized. This method can be called by any user with the burner role.

Parameters:

- `tokenId: uint256` – The ID of the token to burn.

Token Behavior Methods - Transferable Behavior**safeTransferFrom**

This method can be called by any user with tokens to transfer them to another user.

Parameters:

- `from: string` – The wallet address of the sender, which must not be zero.
- `to: string` – The wallet address of the receiver, which must not be zero.
- `id: uint256` – The ID of the token to transfer.

- `value: uint256` – The quantity of tokens to transfer.
- `data: bytes` – Additional data in byte format.

safeBatchTransferFrom

This method can be called by any user with tokens to transfer them in batches to another user.

Parameters:

- `from: string` – The wallet address of the sender, which must not be zero.
- `to: string` – The wallet address of the receiver, which must not be zero.
- `id: uint256[]` – A list of IDs of tokens to transfer.
- `value: uint256[]` – A corresponding list of quantities of tokens to transfer.
- `data: bytes` – Additional data in byte format.

Token Behavior Methods - Delegable Behavior**setApprovalForAll**

This method can be called by any user who owns tokens to grant or revoke permission for an operator to transfer the caller's tokens.

Parameters:

- `operator: string` – The wallet address of the operator, which must not be zero.
- `approved: bool` – A flag indicating whether the operator has permission to transfer the caller's tokens.

isApprovedForAll

This method can be called by any user who owns tokens to check whether a specified user has permission as an operator to transfer the caller's tokens.

Parameters:

- `account: string` – The address of the user who granted or revoked permission for the operation to transfer their tokens.
- `operator: string` – The wallet address of the operator, which must not be zero.

Returns:

- A Boolean value (true or false) indicating whether the specified user has permission as an operator to transfer the caller's tokens.

Token Behavior Methods - Pausable Behavior**paused**

This method checks whether a contract is paused. This method can be called only by any user.

Parameters:

- None

pause

This method pauses a contract. This method can be called only by a Token Admin.

Parameters:

- None

unpause

This method starts a paused contract. This method can be called only by a Token Admin.

Parameters:

- None

Token Behavior Methods - Lockable Behavior**lockNFT**

This method locks an NFT. This method can be called only by a user with the TOKEN_SYS_VAULT_ROLE role.

Parameters:

- tokenId: uint256 – The ID of the token to lock.

isNFTLocked

This method checks whether an NFT is locked. This method can be called only by a user with the TOKEN_SYS_VAULT_ROLE role or by a Token Admin.

Parameters:

- tokenId: uint256 – The ID of the token to check.

Returns:

- A Boolean value (true or false) indicating whether the NFT is locked.

Context Management Methods**setContext**

This method sets the token contract address for the account contract. This method can be called only by a Token Admin.

Parameters:

- tokenAddress: address – The contract address of the deployed token, which must not be zero.

getContext

This method gets the token contract address for the account contract. This method can be called by any user.

Parameters:

- None

Returns

- The token contract address

setGovernanceContext

This method sets the governance contract to use to manage the governance life cycle for the account contract. This method can be called only by a Token Admin.

Parameters:

- `uuid: string` – The unique ID set by the digital assets Token Admin before contract deployment.
- `governance: address` – The contract address of the deployed governance contract, which must not be zero.

getGovernanceContext

This method gets the governance context details. This method can be called by any user.

Parameters:

- None

Returns

- A `governanceContext` object.

Return Value Example:

```
{
  "governanceUUID": "abc-def",
  "governance": "0x00000000",
  "deploymentActive": "true"
}
```

isDeploymentActive

This method checks whether the deployment is active. This method can be called by any user.

Parameters:

- None

Returns:

- A Boolean value (true or false) indicating whether the deployment is active.

5

Wholesale Central Bank Digital Currency Application

The wholesale central bank digital currency (CBDC) sample application and related packages support transaction scenarios between a system owner and participating institutions.

The wholesale CBDC sample represents the life cycle of a wholesale CBDC token. The sample specification file uses the extended Token Taxonomy Framework standard that is supported by Blockchain App Builder. The generated smart contract includes methods from initializing a wholesale CBDC token, account operations, and role assignments to minting, transfers, and burning of CBDC tokens. It also provides notary accounts for approving minting, transfers, and burning operations, and it supports compliance through account-level daily limits and auditing procedures.

The wholesale CBDC solution comprises the following packages.

- Prebuilt wholesale CBDC smart contract
- Wrapper APIs for CBDC setup and lifecycle operations
- Database view definitions to support analytics
- A sample GUI application

To get the wholesale CBDC packages, in the Oracle Blockchain Platform Enterprise Edition console click **Digital Assets**. Click **Applications** and then select **Wholesale CBDC Application**.

To preview sample actions for the application without actually calling a smart contract, click **Lifecycle**.

Quick Start

Complete the following steps in order, setting up the central bank first and then repeating the required setup for each participating financial institution.

1. Deploy the wholesale CBDC application from the Oracle Blockchain Platform instance by using one-click deployment. Wait until the smart contract, wrapper API, rich history services, application, and service configuration are ready.
2. Open the application URL. Enter the wholesale CBDC token ID, review the prefilled values, and optionally update the user-facing customizations. Apply the configuration before creating accounts.
3. Sign in as the bootstrap central bank administrator and initialize the token. Register or verify the founder and participant institution organizations. Create the central bank accounts and a financial institution administrator account for each participating institution, selecting or creating each user's wallet and choosing the appropriate application group and limits.
4. Complete deployment and setup for each participating organization. The financial institution administrator signs in and creates the remaining manager, officer, user, and auditor accounts for the organization.

5. After the financial institution manager accounts are created, as the central bank administrator use the **Account Actions > Sync Roles** command on each manager account to add its Escrow ledger role. The financial institution administrator can assign the Manager application group but cannot grant the required Besu Notary role, which is displayed in the UI as Escrow. Verify all accounts, wallets, groups, roles, and limits.
6. Sign in as each central bank and financial institution persona to verify that the expected workspace opens, the wallet and account context is correct, and that restricted actions are unavailable.
7. Run operations in the token life cycle: request and approve creation, transfer to the issuer, distribute or return value, approve participant transfers, and retire value.
8. Use the **Transactions** page and auditor workspaces to confirm balances, pending requests, operation IDs, and final ledger history.

Install the Wholesale CBDC Application

Before you can use the application, you must configure the rich history database, deploy the application, and run an ORDS script.

The wholesale CBDC application uses Oracle AI Database to store indexed ledger history, with application-specific views so that the data can be queried. Oracle REST Data Services (ORDS) expose the views via authenticated REST routes. You must configure a rich history database profile for Oracle AI Database in Blockchain Platform Manager and then select that profile when you create an instance. For deployments with multiple organizations, each participating organization must have rich history enabled. Then, after the wholesale CBDC and account contracts are deployed, you run the bundled ORDS script against the same rich history database to create the required database views, ORDS REST routes, OAuth client, and optional analytics views.

Configure the Rich History Database Profile for the Wholesale CBDC Application

You use Blockchain Platform Manager to create a profile for the rich history database.

You must meet the following prerequisites:

- Blockchain Platform Manager access to create rich history database profiles and create instances.
- An Oracle AI Database that is reachable from the Blockchain Platform Manager and from the computer where you run the ORDS script. The database must be provisioned.
- A database user account approved by the DBA to connect, read the rich history tables, create or replace views, and administer the ORDS objects that are required by the script. The DBA must grant the required object and ORDS administrative privileges for the selected schema.
- The following information from the DBA: the database endpoint and connection string or TNS service alias. The database username and password. Network approval for Blockchain Platform Manager to reach the database.
- If the database uses a wallet-based TLS connection, an extracted Oracle wallet .zip file, supplied by the DBA.

Do not place the wallet or any other production secrets in source control. Store database passwords, wallet files, OAuth client secrets, and bearer tokens in an approved secret management system.

1. In Blockchain Platform Manager, select **Configuration**.
2. Select **Rich History Database**.
3. Select **Add New**.
4. For **Database Type**, select **Oracle Database**.
5. Enter a unique name for **Config Name**. Use a name that makes the environment and purpose obvious, because this name is required later when you create an instance. For example, use a name of the form <environment>-wcbdc-rich-history-oracle.
6. Enter the database username, password, endpoint, and the connection string supplied by the DBA.
7. If the database connection requires a wallet, upload the wallet package.
8. Do not select **Use Database Blockchain Table** unless specifically required by your DBA.
9. Select **Save**.
10. Validate the profile by confirming that it is displayed when you select **Configuration > Rich History Database** and that the selected type is Oracle Database.

If Blockchain Platform Manager reports a connection failure, correct the database endpoint, connection string, credential, wallet, or network rule before attempting to create an instance.

You can now create the instance and deploy the application.

Create the Instance for the Wholesale CBDC Application

After you configure the rich history database profile, you create the instance for the application.

Do not complete the following steps until you have configured the rich history database profile.

1. In Blockchain Platform Manager, Select **Instances**, then select **Create Instance**.
2. Enter the required settings, including archive node count, member node count, and smart contract governance mode.
3. For **Rich History DB Profile**, select the profile that you created, which must be an **Oracle Database** profile, not an **Internal DB** profile.

For more information about creating an instance, see *Create an Instance in Administering Oracle Blockchain Platform Enterprise Edition for Besu*.

4. To verify that rich history data is available, after the instance is created, deploy and initialize the wholesale CBDC application. For more information, see *One-Click Application Deployment*.
5. Run representative transactions, such as account creation, so that indexed rich history data is generated.

The ORDS Script uses the exact table prefix before the following rich history tables.

```
<BESU_TABLE_PREFIX>_more  
<BESU_TABLE_PREFIX>_store_state  
<BESU_TABLE_PREFIX>_store_hist
```

For example, if the tables are named obpBesu_centralbank_123_more, obpBesu_centralbank_123_store_state, and obpBesu_centralbank_123_store_hist, set

the following prefix. Use the exact database object prefix, including case. Do not supply a view name, a table suffix such as `_more`, or a fabricated deployment label.

```
BESU_TABLE_PREFIX="obpBesu_centralbank_123"
```

You can now prepare and run the ORDS script.

Run the ORDS Script for the Wholesale CBDC Application

After you create the instance and deploy the application, you can run the ORDS script.

You must meet the following prerequisites:

- Node.js version 20.19 or later, 22.13 or later, or 23.5 or later.
- You must have the ORDS script .zip file, which is downloadable from the service console.

Do not place production secrets in source control. Store database passwords, wallet files, OAuth client secrets, and bearer tokens in an approved secret management system. The script's committed .env file is a placeholder template only.

1. Extract the ORDS script .zip file to a working directory.
2. In the `ORDSscript` directory, open a terminal.
3. Run the following commands to confirm that the version of Node.js on your system is supported.

```
node --version
npm install
```

4. Configure the .env file by replacing placeholder values in the provided template. Do not add database passwords to this file; the command-line interface prompts for them interactively.

```
# Common ORDS configuration
CONNECTION_STRING="<Oracle connection string or TNS service alias>"
VIEW_PREFIX="<unique deployment namespace>"
MODULE_NAME="<unique ORDS module name>"
BASE_PATH="<unique/ords/base/path/>"
ITEMS_PER_PAGE="100"
ORDS_REST_BASE_URL="https://<ords-host>"
ALIAS_NAME="<REST-enabled schema alias>"
ROLE_NAME="<unique ORDS role>"
PRIVILEGE_NAME="<unique ORDS privilege>"
LABEL="<privilege label>"
DESCRIPTION="<deployment description>"
CLIENT_NAME="<OAuth client name>"
OWNER="<OAuth client owner>"
SUPPORT_EMAIL="<support email>"

# Besu WCBDC configuration
BESU_TABLE_PREFIX="<exact prefix before _more, _store_state and
_store_hist>"
BESU_TOKEN_CONTRACT="0x<deployed WCBDC token proxy address>"
BESU_ACCOUNT_CONTRACT="0x<deployed Account proxy address>"
```

CONNECTION_STRING

If the `CONNECTION_STRING` parameter is a TNS alias, extract the wallet and configure the Oracle client environment so that the `tnsnames.ora` file can be found. The compressed wallet `.zip` file alone is not sufficient.

ALIAS_NAME

REST-enabled Oracle schema alias used in the public ORDS URL and token URL.

BESU_ACCOUNT_CONTRACT

Deployed account proxy contract address.

BESU_TABLE_PREFIX

Exact rich history table prefix; it determines the source `_more`, `_store_state`, and `_store_hist` tables.

BESU_TOKEN_CONTRACT

Deployed wholesale CBDC token proxy contract address.

MODULE_NAME and BASE_PATH

These parameters define the ORDS module identity and URL path. Keep both deployment-specific and unique.

PATTERN, CHAINCODE_NAME, INSTANCE_NAME, CHANNEL_NAME

These parameters are used only in the Hyperledger Fabric version of the wholesale CBDC application. They do not apply to the Besu version.

VIEW_PREFIX

Unique namespace for operational and optional analytics views. Use a different prefix for each deployment that shares an Oracle schema.

5. Enter the following command to run the script.

```
npm run start
```

- a. Enter the Oracle database username and password.
 - b. Select **Besu** as the blockchain platform.
 - c. Select **WCBDC** as the application type.
 - d. If a prior run was incomplete, review the rollback prompt before choosing to revert objects.
 - e. Select **Yes** to create ORDS REST endpoints.
 - f. Select analytics views only when the deployment requires analytics output.
 - g. Select **Yes** to generate an OAuth client when a new client is required for this deployment.
6. If you selected endpoint and OAuth creation, retain the generated output in an approved secret store. The output contains the callable endpoint map, OAuth client ID, OAuth client secret, and a short-lived bearer-token response. Do not copy the client secret or bearer token to any source control system, file, wiki, or project management software that does not support approved secret handling. The following text shows sample output.

```
{
  "ORDSendpoint": "<generated endpoint URL or endpoint map>",
  "clientId": "<OAuth client ID>",
  "clientSecret": "<OAuth client secret>",
```

```

    "bearerToken": {
      "access_token": "<short-lived access token>",
      "token_type": "bearer",
      "expires_in": 3600
    }
  }
}

```

7. If you selected OAuth client generation, retain the reported client ID and client secret in an approved secret store and use them to call an endpoint to verify system function. The following text shows the form of a token request

```

curl --request POST \
  "https://<ords-host>/ords/<ALIAS_NAME>/oauth/token" \
  --user "<client-id>:<client-secret>" \
  --header "Content-Type: application/x-www-form-urlencoded" \
  --data "grant_type=client_credentials"

```

Endpoint URLs use the following format.

```
https://<ords-host>/ords/<ALIAS_NAME>/<BASE_PATH>/<route>
```

The following command shows an example verification test.

```

curl --location \
  "https://<ords-host>/ords/<ALIAS_NAME>/<BASE_PATH>/getAllCBDCAccounts?token_id=<TOKEN_ID>" \
  --header "Authorization: Bearer <access-token>" \
  --header "Accept: application/json"

```

8. Verify that the script ran successfully by running the following checks.

Blockchain Platform Manager profile

The Oracle AI Database rich history profile exists and is selected for the instance.

Rich history tables

The configured BESU_TABLE_PREFIX resolves to `_more`, `_store_state`, and `_store_hist` tables.

Indexed data

At least one representative wholesale CBDC transaction is shown in the rich history data.

Views

All operational views exist under the configured VIEW_PREFIX.

ORDS module

The configured module is published at the expected BASE_PATH.

OAuth

A client credentials request returns an access token.

API

A valid authenticated request returns HTTP 200. An empty items list is acceptable when no matching ledger data exists.

Isolation

Views, module, role, privilege, client, and alias names are unique for deployments sharing one Oracle database.

The following SQL queries are useful for validation.

```
SELECT view_name
FROM user_views
WHERE view_name LIKE UPPER('<VIEW_PREFIX>') || '_%'
ORDER BY view_name;

SELECT name, uri_prefix, items_per_page, status
FROM user_ords_modules
WHERE name = '<MODULE_NAME>';
```

Wholesale CBDC Deployment Reference

The following information includes descriptions of the database objects that are created by the ORDS script, and troubleshooting tips.

The ORDS script creates the following operational views using the configured VIEW_PREFIX parameter.

```
<VIEW_PREFIX>_ACCOUNTS_V
<VIEW_PREFIX>_TRANSFER_EVENTS_V
<VIEW_PREFIX>_HOLDS_V
<VIEW_PREFIX>_ACTIVE_ROLES_V
<VIEW_PREFIX>_ACCOUNT_TX_HISTORY_V
<VIEW_PREFIX>_ACCOUNT_HISTORY_V
```

If you select analytics, the following analytics views are created, scoped to the deployment.

```
<VIEW_PREFIX>_ACCOUNTS_MOD
<VIEW_PREFIX>_TRANSACTION_MOD
<VIEW_PREFIX>_ACCOUNTS_TRANSACTION_MOD
```

The script also creates or configures the ORDS module, routes, schema alias, role, privilege, OAuth client, and client-role association using the configured values.

View Catalog and Query Logic

The following views are ordinary Oracle views in the selected schema; they do not modify the extractor tables.

View	Primary Source	Purpose/Logic
<VIEW_PREFIX>_ACCOUNTS_V	_more, _store_state	Builds the current account directory from createAccount calls and account/token state. It resolves account, user and organization identity, balances, hold balances, status, daily limits, token metadata, and creation transaction metadata.

View	Primary Source	Purpose/Logic
<VIEW_PREFIX>_TRANSFER_EVE NTS_V	_more	Normalizes token receipt logs into transfer-event rows. It extracts sender, receiver, amount, function, block/transaction metadata, and optional ~OBPRPCTX~.value.category context.
<VIEW_PREFIX>_HOLDS_V	_store_hist, _more, _store_state	Reconstructs mint, burn, and transfer-hold records. It extracts operation/holding ID, accounts, notary, amount, expiry, token, category, description, and the latest lifecycle status. It also provides fallback rows when a current state row is not resolvable.
<VIEW_PREFIX>_ACTIVE_ROLES _V	_more	Applies role add/remove history and returns the latest active role assignment per account and role.
<VIEW_PREFIX>_ACCOUNT_TX_H ISTORY_V	The preceding operational views	Produces account-facing transaction-history rows, including both sides of transfers and normalized mint/burn/hold lifecycle outcomes. It enriches from/to identities from the _ACCOUNTS_V view.
<VIEW_PREFIX>_ACCOUNT_HIST ORY_V	_more, _ACCOUNTS_V	Produces account configuration history from account contract mutations, including daily limit revisions, timestamps, block numbers, and transaction IDs.
<VIEW_PREFIX>_ACCOUNTS_MOD (optional analytics)	_ACCOUNTS_V	Exposes account metadata in the analytics-compatible account shape.
<VIEW_PREFIX>_TRANSACTION_ MOD (optional analytics)	_TRANSFER_EVENTS_V, _HOLDS_V, _store_hist	Exposes normalized business transactions, including transfer, mint, burn, request, hold, release, and resulting balances where available.
<VIEW_PREFIX>_ACCOUNTS_TRA NSACTION_MOD (optional analytics)	_TRANSACTION_MOD, _ACCOUNTS_V	Adds source and destination organization context to analytics transaction records.

ORDS Routes

All routes are GET routes under <BASE_PATH>/ and require an OAuth bearer token. ORDS returns a collection envelope; an empty items array with HTTP 200 means that the request was valid but had no matching indexed records.

Route	Use and Response Behavior	Required Inputs	Optional Inputs and Default Behavior
getCBDCTransactionById	Returns one normalized ledger-transaction detail record.	token_id identifies the WCBDC token; transaction_id is the Besu transaction hash/identifier to retrieve.	None. No matching transaction returns an empty collection.
getCBDCAccountTransactionHistoryWithFilters	Returns history for the account identified by its business identity.	token_id identifies the token; org_id identifies the organization; user_id identifies the account user.	startTime and endTime restrict the time range; omitted means no lower/upper time boundary. bookmark is the zero-based offset; omitted means 0. pageSize is the number of rows requested; omitted uses the handler default.
getAllCBDCAccountTransactionHistoryWithFilters	Returns token-wide account transaction history.	token_id identifies the token.	startTime, endTime, bookmark, and pageSize behave as for the account-specific history route.
getCBDCAccountStatusHistory	Returns indexed status-history rows for a business account.	token_id, org_id, and user_id identify the account.	None. No matching account/history returns an empty collection.
getApproverActionHistory	Returns mint/burn decisions and final transfer-hold actions. executeHold is executed; releaseHold is rejected.	token_id limits results to the token.	caller_account_id is the manager/approver wallet address. Omit it to return actions by all actors; provide it to return only that wallet's actions. It is a data-scope filter, not authorization.
getTokenHistory	Returns token-level history and supply-related information.	token_id identifies the token.	None.
getCBDCAccountsByRole	Returns accounts associated with a role.	role is the role name to match.	None; the result is not token-filtered by this route.
getUsersByRole	Returns user/account role assignments for a role.	role is the role name to match.	None.
getOrgAccountsByRole	Returns accounts in one organization that have a role	org_id identifies the organization; role is the role name.	None.
getAllCBDCAccounts	Returns all indexed accounts for one token.	token_id identifies the token.	None.
getAllActiveCBDCAccounts	Returns indexed accounts whose current status is active.	token_id identifies the token.	None.
getAllSuspendedCBDCAccounts	Returns indexed accounts whose current status is suspended.	token_id identifies the token.	None.

Route	Use and Response Behavior	Required Inputs	Optional Inputs and Default Behavior
getAllOrgAccounts	Returns all indexed accounts in one organization.	org_id identifies the organization.	None; the result is not token-filtered by this route.
getPendingCBDCIssuance	Returns current pending transfer-hold operations only; terminal, executed, rejected, and released operations are excluded.	token_id identifies the token.	caller_account_id is the notary/manager wallet. Omit it for the full pending queue; provide it to return only holds assigned to that wallet.
getPendingCBDCRequest	Returns current pending mint/burn requests only.	token_id identifies the token; request_type selects the supported request type, such as mint or burn.	None. Approved or rejected requests are excluded.
getPendingRequest	Compatibility route for the same pending-request use case.	token_id and request_type have the same meaning as getPendingCBDCRequest	None.
getCBDCAccountsByUser	Returns account records for one organization/user/token combination.	token_id, org_id, and user_id identify the account owner and token.	None.
getCBDCAccountHistory	Returns account configuration/history revisions.	account_address is the blockchain account/wallet address.	offset is the zero-based row offset; omit for the first page. limit is the maximum row count; omit to use the SQL handler default.
getCBDCAccountStatus	Returns the current indexed account status.	token_id, org_id, and user_id identify the account.	None.
getRolesByAccount	Returns roles for a resolved account.	token_id is always required. Supply either account_address (the blockchain account/wallet address), or the combined org_id and user_id business lookup.	If account_address is supplied it is the direct account filter; otherwise the route uses org_id and user_id. If neither lookup resolves an account, the response is empty.
getAllRegisteredOrgs	Returns organizations represented by indexed accounts for a token.	token_id identifies the token.	None.
getCBDCMetadata	Returns indexed immutable token metadata.	token_id identifies the token.	None.
getCBDCRetiredQuantity	Returns retired/burned quantity information for a token.	token_id identifies the token.	None.

Pagination and timestamps: For the two history routes that support the bookmark and pageSize parameters, the bookmark parameter is a zero-based row offset and the pageSize parameter is camel case. Omit either value to use the handler default. The startTime and endTime

parameters, when supplied, use ISO-8601 timestamp text in the format YYYY-MM-DDTHH:MI:SS+/-HH:MI.

Endpoint inventory: The exact route inventory belongs to the script version packaged for the deployment. After the run, use the ORDS OpenAPI catalog or the generated endpoint output as the authoritative list for the client application.

Troubleshooting

Message or Condition	Resolution
NJS-516: no configuration directory set	A TNS alias was used without an accessible wallet/ configuration directory. Extract the wallet and configure the Oracle client so that the <code>tnsnames.ora</code> file is available.
NJS-511 or ORA-12506	Listener, endpoint, service name, network allowlist, or wallet configuration issue. Validate the Oracle connection outside the script with the DBA.
OAuth token request returns 404	Verify the exact alias in <code>/ords/<ALIAS_NAME>/oauth/token</code> . The alias is not necessarily the Oracle username.
Endpoint returns 401	Obtain a fresh token and verify the OAuth client has the role/privilege associated with the configured module base path.
Endpoint returns 404	Verify host, alias, base path, route name, and that the module is published.
ORDS-25001	The endpoint reached ORDS but its SQL failed. Verify the table prefix, contract addresses, source data shape, and required request parameters.
ORA-20049: Cannot alter the URL mapping while the schema is enabled	The chosen ALIAS_NAME conflicts with an enabled schema mapping. Reuse the schema's existing alias, or have the ORDS administrator complete the approved disable/remap/enable procedure before running the script again.
Views contain no rows	Confirm the correct rich history profile was selected during instance creation, the table prefix is exact, contract addresses match the deployed proxies, and transactions have been indexed.
Objects conflict with another deployment	Use a distinct VIEW_PREFIX, module name, base path, role, privilege, OAuth client name, and alias.

One-Click Deploy the Wholesale CBDC Application

One-click deployment prepares the required back end services and supplies the application configuration.

One-click deployment provisions and connects the following application components.

- Besu wholesale CBDC smart contract and deployment metadata, which provides the token operations and contract details that are used by the application.
- Besu wrapper API, which provides the application API for contract and account operations.
- Rich history database objects and scripts, which store transaction and operation history.
- Wholesale CBDC ORDS deployment and configuration, which exposes history and related back end services.

- Wholesale CBDC web application and service configuration, which connects the UI to the deployed back end services.

One-click deployment also prepopulates the **Setup** page with service endpoints, contract addresses, UWaaS and wallet settings, OIDC or LDAP settings, ORDS settings, and the packaged default customizations. Users do not need to enter these back end or identity provider values again. Only the wholesale CBDC token ID is required on the **Setup** page. Optional user-facing customizations can also be changed there.

Before you begin, you need the following information.

- The target Oracle Blockchain Platform instance where the wholesale CBDC application will be deployed.
- The wholesale CBDC token ID, to connect the application to the deployed token contract.
- Bootstrap user details that were supplied during when the smart contract was initialized, which identify the first central bank administrator and provide initial administrative access.
- Optionally, display name, logo, currency, organization, or role label changes, which control the interface that users see after setup.

Complete the following steps for one-click deployment

1. Start the wholesale CBDC one-click deployment in Blockchain Platform Manager for the selected instance.
2. Wait until the smart contract, wrapper API, rich history, ORDS, and application components report a successful or ready state. This confirms that the application dependencies are available.
3. Open the application URL. The /set up page is displayed because deployment has completed the back end setup but the application configuration is not complete yet.
4. Enter the wholesale CBDC token ID and review the prefilled values. Do not replace back end values unless you are making an actual change to the deployment.
5. Optionally, change the user-facing customization values, and then select **Apply Changes**. This validates the required fields, saves the token ID and customization values, combines them with the prefilled deployment configuration, and applies the resulting configuration to the application.

After you select **Apply Changes**, wait for the configuration update or redeployment to complete. The application then returns to the sign-in page. The central bank administrator and financial institution administrator can sign in without a CBDC account if their selected wallet already has the matching Token Admin or Org Admin Besu contract role. Every persona still needs an eligible wallet; other personas also need an active CBDC account. One-click deployment does not create users, wallets, or CBDC accounts.

Configure the Wholesale CBDC Application

You configure the application by checking application and connection settings, configuring the sign-in process, and configuring user interface settings.

The **Setup** page acts differently depending on how the wholesale CBDC application was deployed. With one-click deployment, the installation configuration from Oracle Blockchain Platform Enterprise Edition is used, including the identity provider configured for the instance, and this configuration is used to prefill the application values. With a local deployment, the deployer must deploy or connect to the required services and enter the application configuration manually.

Application and Connection Settings

Review or enter the following settings.

The network selection is fixed by the deployed runtime environment. Do not change prefilled contract addresses, service identifiers, OIDC settings, or ORDS settings unless the replacement services belong to the same deployment and the change is part of an approved deployment procedure.

Setting	One-Click Deployment	Local Deployment
Token ID	Enter the token ID deployed for this application.	Enter the token ID manually.
Protocol and authentication mode	Prefilled from the Oracle Blockchain Platform deployment; normally review only.	Enter the approved local protocol and authentication mode.
Besu wrapper API and access token URL	Prefilled by Blockchain Platform Manager.	Enter the URLs for the locally deployed services.
Oracle Blockchain Platform Instance client ID and client secret	Prefilled from the Oracle Blockchain Platform instance configuration; the secret remains masked.	Enter the local client values when the deployment requires them.
Account and wholesale CBDC contract addresses	Prefilled from the deployed Besu contracts.	Enter the addresses returned by the local contract deployment.
UWaaS wallet URL, organization ID, and wallet service instance ID	Prefilled from the Oracle Blockchain Platform instance setup.	Enter the local UWaaS and wallet service values.
Default caller address	Prefilled when required by the deployment.	Enter it when the local deployment requires a default caller.
ORDS OAuth URL, client ID, and client secret	Prefilled from the ORDS deployment managed by Blockchain Platform Manager; the secret remains masked.	Enter the values for the locally deployed ORDS services.

Sign-In Process

For one-click deployment, the identity provider is the same one configured for the Oracle Blockchain Platform installation. OIDC or LDAP values are prefilled by Blockchain Platform Manager. Review the values, but do not replace these values unless the Oracle Blockchain Platform deployment configuration is intentionally changed.

For a local deployment, no identity provider configuration is inherited automatically. Enter the issuer URL, client ID, client secret if required, scopes, user-ID claim, exact redirect URI, and post-logout redirect URI. Register the same URLs in the selected identity provider.

OIDC authenticates the identity. The application still checks the selected organization, wallet service instance, active CBDC account, application group, and smart contract role before allowing an operation. Configure multi-factor authentication in the identity provider.

Application Presentation

Open the customization cards and review the following presentation settings.

Customization Area	Controls
Token and CBDC account display names	Names shown in headers, account cards, balances, and transaction forms.
Currency and number format	Currency symbol, decimal display, grouping, and symbol position. For U.S. dollars, select USD — US Dollar (\$) and confirm the number preview. Besu token decimals come from the deployed contract and cannot be changed during configuration. If the displayed precision is wrong, correct the contract deployment instead of changing display settings to hide the mismatch. The token's contract-defined decimal precision is read-only in Setup and is not the same as the currency symbol and number formatting used for display.
Login branding	Message and image shown before sign-in.
Application background	Background color used across the application shell.
Transaction labels	User-facing names for transaction types.
Organization display names, logos, and order	Presentation of organizations in dropdowns and workspace navigation. Live UWaaS IDs, names, sequences, and founder tags remain authoritative.
Application group labels	Names shown for central bank and financial institution personas.
Ledger role labels	Names shown for smart-contract permissions such as Token Admin, Minter, Burner, and Escrow/Notary.

Apply the Configuration

For one-click deployment, the **Apply Changes** command validates required fields, saves the token ID and optional customizations, combines them with the configuration from Blockchain Platform Manager, and applies the resulting configuration to the application. When the configuration update or redeployment is complete, the application returns to the sign-in page.

For a local deployment, the **Apply Changes** command validates required fields and writes the local configuration to the `.env.local` file. Masked secret fields can be left blank to retain their current values. You must restart the application when requested. The application then returns to the sign-in page.

Wholesale CBDC Sample Application Workflow

After you install and configure the sample wholesale CBDC application, you can use it in scenarios where a system owner (a central bank) and participant organizations (other financial institutions) interact in an interbank market.

The sample application supports eleven roles, or personas. Each role has a different interface and set of operations that support the entire workflow of token management in the wholesale CBDC scenario.

Central Bank Roles

- Central bank administrator: Configures tokens and organizations; creates wallets and CBDC accounts; sets limits, purposes, groups, and roles.

- Central bank creator: Requests token creation and transfers the created value to a central bank issuer.
- Central bank issuer: Issues value to participants and returns value for retirement..
- Central bank manager: Approves or rejects creation, issuance, transfer, and retirement requests.
- Central bank retiree: Submits retirement or burn requests.
- Central bank auditor: Reviews network activity, balances, pending requests, and history without changing data.

Financial Institution Roles

- Financial institution administrator: Manages participant wallets, accounts, application groups, status, and limits.
- Financial institution manager: Approves or rejects transfers for the participant organization.
- Financial institution officer: Deposits value to the central bank and transfers value to permitted users.
- Financial institution user: Transfers value to permitted recipients.
- Financial institution auditor: Reviews participant/organization activity without changing data.

Onboarding

After you install and configure the application, you prepare users, wallets, CBDC accounts, and ledger roles. Complete the following steps before you run token operations.

1. The central bank administrator logs in first. Select the bootstrap wallet that was granted the Token Admin Besu role when the smart contract was initialized. This is the initial administrative entry point even before a CBDC account exists.
2. The central bank administrator verifies the organization registry. Register or review the founder and participant organizations, confirm the founder sequence, and verify the organization order and display values. For more information, see [Central Bank Administrator](#).
3. The central bank administrator initializes the token and creates the initial accounts. Create wallets and CBDC accounts for central bank personas and for the financial institution administrators who will administer participating organizations. Select the correct wallet, application group, and account limits.
4. Each participating organization is deployed and configured. Use the same one-click or local deployment process for the participant application. One-click deployment uses the Oracle Blockchain Platform configuration; local deployment requires manual configuration.
5. The financial institution administrator logs in for the participant organization. Load or create the organization's wallets, then create manager, officer, user, and auditor accounts using the permitted application groups and limits.
6. The central bank administrator synchronizes manager roles. After each financial institution manager account is created, use the **Account Actions > Sync Roles** command on that account to apply its Escrow ledger role. Application groups control workspace access; ledger roles control contract operations.
7. Each persona logs in and verifies access. Select the intended wallet when more than one is available, confirm the correct workspace opens, and verify that permitted and restricted actions match the assigned application group and ledger role. Use the persona sequence in the following table.

8. After all persona configuration and access checks are complete, follow the application workflow to work with the wholesale CBDC token life cycle.

Persona Sequence

Sequence	Persona	Responsibility	Information
1	Central bank administrator	Completes central bank, organization, wallet, account, and role setup.	Central Bank Administrator
2	Financial institution administrator	Sets up participant wallets and accounts.	Financial Institution Administrator
3	Central bank creator	Submit requests to create tokens.	Central Bank Creator
4	Central bank manager	Approves or rejects creation requests.	Central Bank Manager
5	Central bank issuer	Issues value to participants and returns value for retirement.	Central Bank Issuer
6	Financial institution manager	Approves or rejects participant transfers.	Financial Institution Manager
7	Financial institution officer	Deposits and transfers value.	Financial Institution Officer
8	Financial institution user	Transfers value to permitted recipients.	Financial Institution User
9	Central bank retiree	Submits retirement requests after value is returned.	Central Bank Retirer
10	Central bank auditor	Reviews central bank activity and history.	Central Bank Auditor
11	Financial institution auditor	Review participant organization activity and history.	Financial Institution Auditor

Application Workflow (Token Life Cycle)

After deployment and setup, the personas in the scenario engage in the following tasks as part of the complete token life cycle.

1. The central bank creator requests token creation. The request remains pending and does not yet credit the creator.
2. The central bank manager approves or rejects the creation request. Approval credits the creator; rejection does not.
3. The central bank creator transfers approved value directly to the central bank issuer. This transfer has no additional approval step.
4. The central bank issuer transfers directly to the central bank retiree to return value for retirement, or submits a distribution to a financial institution officer. The officer distribution is held for central bank manager approval; rejection releases the hold.
5. The financial institution officer transfers to the central bank issuer, another financial institution officer, or a financial institution user. Each officer transfer is held for financial institution manager approval; rejection releases the hold.
6. A financial institution user transfers to a permitted financial institution officer or user. Each user transfer is held for financial institution manager approval; rejection releases the hold.

7. The central bank retiree submits a burn request after receiving value. The central bank manager approves or rejects the request. Approval removes the value from circulation; rejection leaves the burn incomplete.
8. The central bank auditor reviews central bank and financial institution activity available to its audit scope. The financial institution auditor reviews their own institution's activity. Both use operation IDs and final history states to reconcile the sequence.

After each request or decision, refresh the sender, recipient, manager, and auditor views as applicable. Confirm the final status and balance before submitting another operation.

Central Bank Administrator

The central bank administrator has the first operational workspace after deployment.

The central bank administrator initializes the token, registers organizations, creates wallets and CBDC accounts, maintains roles and limits, reviews activity, and configures the application presentation.

Log in as the central bank administrator user with the wallet granted the `Token Admin Besu` contract role when the smart contract was initialized. The wallet must be eligible for the configured organization and wallet service instance. This user can log in before a CBDC account is created. Create the remaining users' wallets and CBDC accounts before they log in.

Before you begin the administrator tasks, the application must be deployed, either by one-click deployment or local deployment, and configured so that it is connected to the Besu contracts, wrapper API service, wallet service, and other supporting services. You must also have the following information.

- Central bank administrator login details from smart contract initialization, to open the application and complete administrative setup before an operational CBDC account exists.
- The token ID and token description supplied for the deployment. The token description is handled during token initialization; account creation uses the selected token ID and account details.
- Organization names, unique organization IDs, and the intended founder/participant order. Organizations must be registered before accounts can be created for them.
- User IDs and eligible wallets for the users who will operate the application. Each operational account is linked to a user and an active wallet.
- The application group and daily limits required for each user. The selected group controls the workspace and the configured limits control account usage.

Typically the central bank administrator completes tasks in the following order.

1. Open the system dashboard and confirm the organization, user, token, and current account totals.
2. Initialize the token if deployment has not already initialized it.
3. Register the founder and participant organizations, and then confirm their display names and sequence order.
4. Create wallets and CBDC accounts for the central bank administrator and other central bank personas.
5. Create accounts for participating organization administrators. After participating organization administrators create their manager accounts, use the **Sync Roles** command on those accounts to add the escrow ledger role.

6. Review or update token details and configure the transaction purposes before operational token transfers begin.
7. Review the account list, organization cards, KPIs, and analytics dashboard.
8. Configure the application settings and customize the application presentation.

The following tasks are typical ones that a central bank administrator completes.

Initialize Token

Initializing the token is the first task that the central bank administrator must complete.

1. Select **Token Management > Initialize Token**. The Initialize Token window opens.
2. Confirm that the token ID is correct.
3. Edit the customer-facing token description as needed.
4. Select **Initialize Token** to complete the process.

Register Organizations

You must register organizations before you create accounts for them. Use the organization sequence to control the order in which organizations are displayed in the application.

1. Select **Participants > Organizations**. The Manage Organizations window opens.
2. Enter an organization name, organization ID (the unique ID used by the network registry), and a sequence number. The founder organization is always first in the sequence. If the requested sequence is already in use, the selected organization and later entries move to preserve a valid order. The application also reconciles missing, duplicate, or non-consecutive values using the registry data and creation order. If the live registry contains more than one founder, correct the founder tags in the registry before continuing.
3. Select **Register Organization**.

Create Wallets and CBDC Accounts

Complete the following steps to create accounts.

1. Select **Participants > Create Bank Account**. The Create Token Bank Account window opens.
 2. Select the **Bank Name**, and enter the **Bank User ID**.
 3. Select **Load wallets** for the configured wallet service instance.
 4. Select the user's wallet, or if the user has no eligible wallet, create a wallet. To create a wallet, select **Create new wallet**. Loading or creating wallets for another organization might require that organization's wallet service instance ID.
 5. Select the **Group** and set the account limits.
- The following table describes the account information fields.

Field	Description
Token ID	The ID of the token that was initialized for the system.
Bank Name	A list of all registered organizations in the network.
Org ID	The organization ID of the bank, which is determined by the selected Bank Name .
Bank User ID	A user ID for the account.

Field	Description
Role	A list of supported roles. Role information is supplied by the selected application group. Roles are assigned by the application during account creation.
Group	A list of application groups (personas).
Daily Maximum Amount	The maximum amount of tokens that can be transacted per day.
Daily Maximum Transactions	The maximum number of transactions allowed per day.
Wallet	Select an active wallet with no existing token account. Use Load wallets to retrieve eligible wallets or Create new wallet when the deployment permits wallet creation.

6. Select **Create Account**. The account details are displayed.

The selected application group and its associated roles are assigned automatically, with the exception of a financial institution manager. After creating a participating institution manager account, open its account actions menu and select **Sync Roles** to apply the required escrow ledger role. Application groups determine the workspace shown in the UI; ledger roles determine contract operations.

Additional Administrator Tasks

Select **Token Management > Manage Purposes** to add, edit, or remove the purpose values that are available for transfers. Removing a purpose does not rewrite existing transaction history.

Select **Insights > Analytics Dashboard** to open the read-only dashboard. From the summary cards, select the **Accounts**, **Transactions**, or **Token** view. Choose an organization and status for a narrower set of results. Select **Refresh Analytics** when the page was opened before a recent operation completed. A "No analytics data" message means the selected view has no available records or is still loading; adjust the filters or refresh the page.

Select **Account > My Account** to open a read-only page for the currently logged-in user. You can use this page to confirm the account selected at sign-in, its organization and token, its status and balance, its limits, and the changes recorded for the account.

Troubleshooting

If organization or account data is unavailable, verify the UWaaS, wrapper API, smart contract, wallet, and ORDS configuration. If a role is missing, use the **Sync Roles** command and ask the user to log out and log in again so that the session is rebuilt.

Central Bank Auditor

The central bank auditor has a read-only workspace for reviewing account activity, transaction history, and audit policy conditions across the central bank and participating organizations.

The central bank auditor is a read-only account. The auditor cannot create or update accounts, move value, approve requests, synchronize roles, or change setup.

Before you can complete central bank auditor tasks, you must have an eligible active CBDC account linked to your wallet. You must be a member of the Central Bank Auditor application group with its assigned ledger role. Finally, there must be transaction data stored in the configured rich history service.

Typically the central bank auditor completes tasks in the following order.

1. Confirm the auditor identity, organization, token, and account status.
2. Review the organization and account status cards.
3. Enable any auditing policies that are required for review.
4. Select the relevant organization or account history control.
5. Apply transaction filters, refresh the history, and sort or paginate the results.
6. Use the transaction ID, timestamp, amount, counterparty, and outcome to reconcile the record.
7. Export the visible history when an offline review is required.

Audit Policies

Audit policies lets the auditor define review criteria for unusual or important activity. Audit policies do not change ledger data and do not approve or reject transactions. You must enable a policy before you can enter or edit the policy parameters.

Policy	Parameters
Single transfer amount	Choose a comparison such as $\geq$, $\leq$, or $=$, and then enter an amount.
Number of transfers exceeding	Enter a count, and then choose the time window: Day , An Hour , Weekly , or Monthly .
Daily transfer total amount exceeding	Enter the amount threshold.
Transfers to specific accounts	Select the accounts.
Transfers from specific accounts	Select the accounts.

Multiple complete, enabled audit policies work together to narrow the displayed history: a record must satisfy each applicable condition. These are UI review criteria, not persistent ledger rules.

Transaction Filters

After you load records from an organization or account history, you can use the following filters to find specific transactions.

Filter	Parameters
Transaction ID	Enter an operation or transaction identifier.
Amount	Select a comparison and enter an amount.
Balance	Select a comparison and enter a balance.
Transaction Type	Select a transaction type.
Initiator	Enter a user or organization value.
Account ID	Enter an account identifier.
Purpose/Memo	Enter the text to search for.
Start Date / End Date	Select the beginning and ending dates.

Selecting a transaction history row opens a read-only Transaction details window, which includes the following fields.

Section	Information Displayed
Summary	Transaction ID, Holding ID, Purpose/Memo, Block Number, Transaction Time, Transaction Type, Amount, and Token ID.
From	Organization, User ID, and From Account ID. The From party displays - for a creation transaction.
To	Organization, User ID, and To Account ID. The To party displays - for a retirement transaction.

Troubleshooting

If data is missing, select **Refresh History**, clear restrictive filters, verify the date range, and confirm the history service and transaction ID. If an unexpected organization or action is visible, verify the account's assigned application group and permissions.

Central Bank Creator

The central bank creator requests new wholesale CBDC supply and transfers available value to eligible central bank recipients.

The central bank creator can request new supply and transfer available value to permitted recipients. The creator cannot approve their own creation requests, change account limits, assign roles, or change setup.

Before you can complete central bank creator tasks, you must have an eligible active CBDC account linked to your wallet. You must be a member of the Central Bank Creator application group with assigned minter role. There must also be a user who is an eligible manager approver with the escrow role, and a user with an eligible recipient account for direct transfers.

Typically the central bank creator completes tasks in the following order.

1. Confirm the creator account is active and review the available balance.
2. Select **Create Tokens**, enter the quantity, and optionally add a memo. If several eligible managers are available, choose the intended approver; if only one is available, the application uses that person automatically or shows the only available value.
3. Submit the request and record its request or transaction ID.
4. Wait for the central bank manager to approve or reject the request.
5. After approval, refresh the creator page and confirm that the balance and history are updated.
6. Select **Issue** beside the intended eligible recipient, confirm the recipient user, optionally enable and select a purpose, enter the quantity, and submit the direct transfer.
7. Refresh history and confirm the transfer status and transaction ID.

The following tasks are typical ones that a central bank creator completes.

Request Token Creation

1. Select **Create Tokens**. The Create CBDC Token Request window is displayed.
2. Enter a **Quantity** and an optional **Memo**. If multiple eligible managers are available, select an **Approver User ID**.
3. Select **Request Mint**. The request is sent for manager approval.

Issue Tokens

1. Select **Issue** on an eligible recipient card. The Transfer window is displayed.
2. Verify the recipient organization for **Transfer Token To** and the recipient for **To User ID**, and enter a **Quantity**. Optionally, select **Purpose** and then select one of the configured purposes.
3. Select **Transfer**. The transaction runs and value is sent directly to the specified recipient.

Transaction Filters

After you load records from an organization or account history, you can use the following filters to find specific transactions.

Filter	Parameters
Transaction ID	Enter an operation or transaction identifier.
Amount	Select a comparison and enter an amount.
Balance	Select a comparison and enter a balance.
Transaction Type	Select a transaction type.
Initiator	Enter a user or organization value.
Account ID	Enter an account identifier.
Purpose/Memo	Enter the text to search for.
Start Date / End Date	Select the beginning and ending dates.

Selecting a transaction history row opens a read-only Transaction details window, which includes the following fields.

Section	Information Displayed
Summary	Transaction ID, Holding ID, Purpose/Memo, Block Number, Transaction Time, Transaction Type, Amount, and Token ID.
From	Organization, User ID, and From Account ID. The From party displays - for a creation transaction.
To	Organization, User ID, and To Account ID. The To party displays - for a retirement transaction.

Troubleshooting

If no approver is available, ask the central bank administrator to verify active manager accounts and escrow roles. If an amount is rejected, check the positive value, decimal precision, balance, and limits. If a request remains pending, give the request ID to the selected manager. If the account is inactive, ask the central bank administrator to review its status.

Central Bank Issuer

The central bank issuer sends wholesale CBDC to eligible participating organization officers and returns value to an eligible central bank retiree account.

The central bank issuer can initiate participant issuance and retirement transfers shown by the application. Issuers cannot approve their own participant issuance, create accounts, assign roles, change limits, or change setup.

Before you can complete central bank issuer tasks, you must have an eligible active CBDC account linked to your wallet, with an available balance. You must be a member of the Central Bank Issuer application group. There must also be active officer accounts in participating organizations, an active central bank retiree account, and an eligible central bank manager approver.

Typically the central bank issuer completes tasks in the following order.

1. Confirm the issuer account is active and review pending and available balances.
2. Expand the participant organization and select an active officer.
3. Select **Issue**, review the central bank manager approval context, optionally enable and select a purpose, enter the quantity, and submit.
4. Record the operation ID and wait for central bank manager to approve or reject the issuance.
5. Refresh history and confirm the issuance status and balance after approval or rejection.
6. For retirement, select an active retiree account, optionally enable and select a purpose, enter the quantity, and submit the **Retirement** transfer.
7. Refresh history and confirm the retirement result.

The following tasks are typical ones that a central bank issuer completes.

Issue Value to a Financial Institution Officer

1. Expand a participant organization, choose an active officer, and select **Issue**. The Transfer window is displayed.
2. Enter a **Quantity**. Optionally, select **Purpose** and then select one of the configured purposes.
3. Select **Transfer**. The request is sent for manager approval.

Return Value for Retirement

1. Select **Retirement** for an active eligible retiree account. Retiree eligibility is based on account and role data, not the displayed name. The Transfer window is displayed.
2. Enter a **Quantity**. Optionally, select **Purpose** and then select one of the configured purposes.
3. Select **Transfer**. The transaction runs.

Transaction Filters

After you load records from an organization or account history, you can use the following filters to find specific transactions.

Filter	Parameters
Transaction ID	Enter an operation or transaction identifier.
Amount	Select a comparison and enter an amount.
Balance	Select a comparison and enter a balance.
Transaction Type	Select a transaction type.
Initiator	Enter a user or organization value.
Account ID	Enter an account identifier.
Purpose/Memo	Enter the text to search for.
Start Date / End Date	Select the beginning and ending dates.

Selecting a transaction history row opens a read-only Transaction details window, which includes the following fields.

Section	Information Displayed
Summary	Transaction ID, Holding ID, Purpose/Memo, Block Number, Transaction Time, Transaction Type, Amount, and Token ID.
From	Organization, User ID, and From Account ID. The From party displays - for a creation transaction.
To	Organization, User ID, and To Account ID. The To party displays - for a retirement transaction.

Troubleshooting

If no participant officers are displayed, ask the participant administrator to create and activate officer accounts. If no retiree is displayed, ask the central bank administrator to verify the account and assigned role. If an amount fails validation, check the balance, daily limits, recipient status, and decimal precision. If an issuance remains pending, give the operation ID to the selected manager.

Central Bank Manager

The central bank manager reviews and approves or rejects pending creation, issuance, and retirement operations that are assigned to them.

The central bank manager can approve or reject assigned pending operations. They cannot create accounts, change limits, assign roles, edit token configuration, or change setup.

Before you can complete central bank manager tasks, you must have an eligible active CBDC account linked to your wallet, with an available balance. You must be a member of the Central Bank Manager application group and have the escrow ledger role. There must be pending requests assigned to you or your organization.

Typically the central bank manager completes tasks in the following order.

1. Review pending operations and the pending totals by organization.
2. Open the page that matches the request: **Issuance, Creation, or Retirement.**
3. Read the operation ID, parties, amount, purpose or memo, and request type.
4. Approve or reject a request, or choose multiple eligible requests and select **Approve Selected**. (You cannot reject multiple requests at a time.)
5. Refresh data and confirm that the request leaves the **Pending** page.
6. Select **History** and verify the terminal status and affected amount.

Troubleshooting

If an action is disabled, verify that the manager account is active and that the row is still pending. If a request is removed, check the **History**, because it may already be in a terminal state. If an amount or party is missing, refresh the data and verify the source operation. If a request is assigned elsewhere, the central bank administrator must verify the approver organization, user, and escrow role.

Central Bank Retirer

The central bank retirer receives value returned by the issuer and submits requests to remove that value from circulation.

The central bank retirer can send burn requests and return available value to eligible issuers. They cannot approve their own retirement request, create accounts, assign roles, change limits, or change setup.

Before you can complete central bank retirer tasks, you must have an eligible active CBDC account linked to your wallet, with an available balance. You must be a member of the Central Bank Retirer application group with the burner ledger role. There must also be value (tokens) returned by a central bank issuer, an eligible manager approver, and an active central bank issuer account.

Typically the central bank retirer completes tasks in the following order.

1. Confirm the retirer account is active and review the total retired and balance amounts.
2. If value has not arrived, ask the central bank issuer to use **Transfer** to send value to this account.
3. Select **Retire Tokens**, choose the manager approver, enter the quantity, and optionally add a memo.
4. Submit the burn request and record the request ID.
5. Wait for the central bank manager to approve or reject the request.
6. Refresh history and confirm the retirement status, balance, and total retired value.

The following tasks are typical ones that a central bank retirer completes.

Retire Tokens

1. Select **Retire Tokens**. The Request Burn Tokens window is displayed.
2. Enter a **Quantity** and an optional **Memo**. If multiple eligible managers are available, select an **Approver User ID**.
3. Select **Request Burn**. The request is sent for manager approval.

Return Value to Issuer

1. Select **Transfer** for an eligible central bank issuer.
2. Select the **To User ID**, enter a **Quantity**. Optionally, select **Purpose** and then select one of the configured purposes.
3. Select **Transfer**. The transaction runs.

Transaction Filters

After you load records from an organization or account history, you can use the following filters to find specific transactions.

Filter	Parameters
Transaction ID	Enter an operation or transaction identifier.
Amount	Select a comparison and enter an amount.
Balance	Select a comparison and enter a balance.

Filter	Parameters
Transaction Type	Select a transaction type.
Initiator	Enter a user or organization value.
Account ID	Enter an account identifier.
Purpose/Memo	Enter the text to search for.
Start Date / End Date	Select the beginning and ending dates.

Selecting a transaction history row opens a read-only Transaction details window, which includes the following fields.

Section	Information Displayed
Summary	Transaction ID, Holding ID, Purpose/Memo, Block Number, Transaction Time, Transaction Type, Amount, and Token ID.
From	Organization, User ID, and From Account ID. The From party displays - for a creation transaction.
To	Organization, User ID, and To Account ID. The To party displays - for a retirement transaction.

Troubleshooting

If no returned value is available, ask the central bank issuer to transfer it. If no approver is available, ask the central bank administrator to verify active manager accounts and escrow roles. If a quantity is rejected, check the balance, limits, and decimal precision. If the account is inactive, ask the central bank administrator to review the account.

Financial Institution Administrator

The financial institution administrator manages wallets and CBDC accounts for a participating organization..

A participant account created with the Org Admin application group adds the corresponding Org Admin role automatically. Financial institution administrators cannot grant minter, burner, token administrator, or central bank escrow roles. The central bank administrator must use the **Sync Roles** to apply the escrow role to participating organization manager accounts.

Before you can complete financial institution administrator tasks, you must be a member of the Financial Institution Admin application group for the selected organization. A CBDC account is not required to log in as a financial institution administrator if the selected eligible wallet already has the Org Admin Besu contract role. The organization and wallet service instance must be registered before creating participant accounts.

Typically the financial institution administrator completes the following tasks.

- Loads or creates wallets
- Creates accounts for organization users, selecting the institution application group that matches the user's intended persona and responsibilities. Org Admin and Org Auditor roles are added for their corresponding groups. The bank administrator must use the **Sync Roles** command for the manager (Escrow) role.
- Edits limits and account states
- Activates or suspends accounts

- Inspects application groups and contacts the central bank administrator when a ledger role must be synchronized.

The following tasks are typical ones that a financial institution administrator completes.

Create Wallets and CBDC Accounts

Complete the following steps to create accounts.

1. Select **Create Bank Account**. The Create Token Bank Account window opens.
2. Enter the **Bank User ID** and confirm the organization and token IDs.
3. Select **Load wallets** and choose the user's wallet, or select **Create new wallet**.
4. Select the **Group** and set the account limits.
The following table describes the account information fields.

Field	Description
Token ID	The ID of the token that selected for the organization.
Organization	The selected organization name.
Organization ID	The registry identifier of the bank.
Bank User ID	The user ID registered for the participant organization.
Role	A list of supported roles. Role information is supplied by the selected application group.
Group	A list of application groups (personas).
Daily Maximum Amount	The maximum amount of tokens that can be transacted per day.
Daily Maximum Transactions	The maximum number of transactions allowed per day.
Wallet	Select an active wallet with no existing token account. Use Load wallets after entering the user and organization to retrieve eligible wallets or select Create new wallet .

5. Select **Create Account**. The account details are displayed.

Troubleshooting

If there is a wallet mismatch, verify the user, organization, and wallet service instance. If a role is missing after account creation, request that the central bank administrator run the **Sync Roles** command. If there is a duplicate or ambiguous account, verify the selected wallet and account address before retrying.

Financial Institution Auditor

The financial institution auditor has a read-only workspace for reviewing account activity, transaction history, and audit policy conditions for a single financial institution.

The financial institution auditor is a read-only account. The auditor cannot create or update accounts, transfer value, approve requests, synchronize roles, or change configuration.

Before you can complete financial institution auditor tasks, you must have an active CBDC account. You must be a member of the Financial Institution Auditor application group for the organization.

Typically the financial institution auditor completes tasks in the following order.

1. Expand the organization's accounts.
2. Select an account.
3. Review or filter the account's transactions.
4. Open transaction details when available.
5. Reconcile pending and terminal states by operation ID.

Audit Policies

Audit policies lets the auditor define review criteria for unusual or important activity. Audit policies do not change ledger data and do not approve or reject transactions. You must enable a policy before you can enter or edit the policy parameters.

Policy	Parameters
Single transfer amount	Choose a comparison such as $\geq$, $\leq$, or $=$, and then enter an amount.
Number of transfers exceeding	Enter a count, and then choose the time window: Day , An Hour , Weekly , or Monthly .
Daily transfer total amount exceeding	Enter the amount threshold.
Any transfers to specific accounts	Select the accounts.
Any transfers from specific accounts	Select the accounts.

Transaction Filters

After you load records from an organization or account history, you can use the following filters to find specific transactions.

Filter	Parameters
Transaction ID	Enter an operation or transaction identifier.
Amount	Select a comparison and enter an amount.
Balance	Select a comparison and enter a balance.
Transaction Type	Select a transaction type.
Initiator	Enter a user or organization value.
Account ID	Enter an account identifier.
Purpose/Memo	Enter the text to search for.
Start Date / End Date	Select the beginning and ending dates.

Selecting a transaction history row opens a read-only Transaction details window, which includes the following fields.

Section	Information Displayed
Summary	Transaction ID , Holding ID , Purpose/Memo , Block Number , Transaction Time , Transaction Type , Amount , and Token ID .
From	Organization , User ID , and From Account ID . The From party displays - for a creation transaction.
To	Organization , User ID , and To Account ID . The To party displays - for a retirement transaction.

Troubleshooting

If data is inconsistent, refresh the history. If records outside the assigned organization are displayed, report an access control issue. If expected history is not displayed, ask the platform administrator to verify ORDS routes and credentials.

Financial Institution Manager

The financial institution manager reviews transfers submitted by users in the organization under the organization's manager/escrow role.

Transfers that are submitted by a financial institution officer or financial institution user remain pending until the manager approves or rejects them. Financial institution managers do not approve distributions from issuers to officers. These transactions are approved by central bank managers.

Before you can complete financial institution manager tasks, you must be a member of the Financial Institution Manager application group for the organization with the required escrow ledger role.

Typically the financial institution manager completes tasks in the following order.

1. Select **Pending Transfers**.
2. Review the source, destination, amount, purpose, and operation ID.
3. Approve or reject an individual transfer, or choose multiple eligible requests to approve them in bulk. (You cannot reject multiple requests at a time.)
4. Refresh and confirm the result in the action history.

Troubleshooting

If encounter a problem or error, confirm that the account is active, the request is still pending, and the current manager is authorized for the request. If a row is no longer displayed, check the history before asking the sender to submit the request again.

Financial Institution Officer

The financial institution officer deposits wholesale CBDC value to the central bank issuer and transfers value to permitted participant officers and users.

Transfer actions to participants must be approved by the financial institution manager.

Before you can complete financial institution officer tasks, you must have an active Financial Institution Officer account with an available balance. Recipient and approver accounts must be active.

The following tasks are typical ones that a financial institution officer completes.

Deposit to the Central Bank

1. Select **Deposit to Central Bank Name**, which opens the transfer window for an eligible bank issuer.
2. Select an active central bank issuer and then select **Transfer**.
3. Enter a **Quantity** to deposit. Optionally, select **Purpose** and then select one of the configured purposes.

4. Review the approver or escrow context displayed by the form. When exactly one eligible manager is available, the application uses that person automatically. Select a value if multiple managers are available.
5. Select **Transfer**, and then record the operation ID.

The request is placed on hold and appears in the financial institution manager's **Pending Transfers** list. The central bank issuer receives the value only after the financial institution manager approves the request.

Transfer to a Recipient

1. Select an active officer or user.
2. Enter a **Quantity** within the available balance and limits. Optionally, select **Purpose** and then select one of the configured purposes.
3. Review the approver or escrow context displayed by the form. When exactly one eligible manager is available, the application uses that person automatically. Select a value if multiple managers are available.
4. Select **Transfer**, and then record the operation ID.

The application validates status, recipient, precision, available balance, daily limits, and approver context. Participant transfers are held for approval by the assigned financial institution manager. After approval, the transfer appears as completed in history and the sender and recipient balances update. If the transaction is rejected, the hold is released and the transfer does not settle.

Troubleshooting

If an account becomes inactive, refresh the recipients. For limit or balance errors, reduce the amount or ask the organization administrator to review the account. Do not resubmit a pending operation.

Financial Institution User

The financial institution user transfers wholesale CBDC tokens to permitted participant officers and users and reviews personal transaction history.

All transfers are routed to the financial institution manager for approval.

The financial institution user cannot create accounts, change limits or roles, approve transfers, initialize the token, or change setup.

Before you can complete financial institution user tasks, you must have an active Financial Institution User account with an eligible selected wallet and an available balance. There must be at least one active recipient.

The following tasks are typical ones that a financial institution user completes.

Transfer Tokens

1. Select an active recipient and then select **Transfer**.
2. Enter the **Quantity** to transfer. Optionally, select **Purpose** and then select one of the configured purposes.
3. Review the approver or escrow context displayed by the form. When exactly one eligible manager is available, the application uses that person automatically. Select a value if multiple managers are available.
4. Review the generated operation ID, and then select **Transfer**.

5. Wait for the financial institution manager to approve or reject the request. The request is shown as pending.
6. Refresh the page and check the final status in transaction history. Approval executes the held transfer and updates the sender and recipient balances after refresh. Rejection releases the hold and records the rejected result in history.

Troubleshooting

For inactive recipient, precision, balance, or limit errors, correct the indicated field or contact the organization administrator. For a pending request, contact the assigned manager. Do not submit a duplicate request.

Troubleshoot the Wholesale CBDC Application

The following information covers how to troubleshoot problems you might encounter with the application.

Sign-In Returns to an Error Page

For OIDC, verify the issuer, client, exact callback URL, audience, discovery/JWKS reachability, TLS trust, and session configuration. For LDAP, verify the LDAP configuration. If accounts are missing, make sure to preserve the Token Admin and Org Admin role-only administrator sign-in exceptions. Confirm that the user has an eligible active wallet/account for the configured organization and wallet service instance.

No Wallet Is Available

Verify UWaaS reachability, organization mapping, wallet service instance, user identity, and access scope. If multiple wallets are returned, select the wallet for the intended wholesale CBDC account.

Account Exists but an Operation Is Unavailable

Check the account status, application group, and ledger role. The central bank administrator can use the **Account Actions > Sync Roles** command when authorized. Log out and log in again after a role change, so that the session is rebuilt.

Organization Order Is Incorrect

Reload the live registry and check for founder tag conflicts. Check for missing, duplicate, or non-consecutive sequence values, which the application can reconcile for ordering. Display name does not control order. Multiple founders require manual data repair before the wholesale CBDC application can reconcile the registry.

Stale Balance or Pending State

Use the page refresh action and check the operation ID in the history. Confirm Besu and ORDS health before retrying. Do not submit the operation again until the first operation is fails or is in a terminal state.

Approve/Reject Is Unavailable

Confirm that the account is active, the user has the required escrow/manager role, the request is still pending, and the request is assigned to the current approval context.

6

Deposit Token Application

The deposit token application scenario supports issuance and distribution of tokens with minimal governance overhead.

The deposit token sample represents the life cycle of a fungible token that represents a bank deposit. All token transfers that are submitted are evaluated against the active approval policy. A transfer settles when approval is not required, otherwise it remains pending and is routed to the configured approver sequence.

The deposit token solution comprises the following downloadable packages.

- Prebuilt deposit token smart contract
- Wrapper APIs for deposit token setup and lifecycle operations
- Database view definitions to support analytics
- Sample web application for working with tokenized deposits

To get the deposit token packages, in the Oracle Blockchain Platform Enterprise Edition console click **Digital Assets**. Click **Applications** and then select **Deposit Token Application**.

To preview sample actions for the application without actually calling a smart contract, click **Lifecycle**.

Quick Start

Complete the following steps in order, setting up the bank first and then repeating the required setup for each participating institution.

1. Deploy the deposit token application from Oracle Blockchain Platform instance by using one-click deployment. Wait until the smart contract, wrapper API, rich history services, application, and service configuration are ready.
2. Open the application URL. Enter the deposit token ID, review the prefilled values, and optionally update the user-facing customizations. Apply the configuration before creating accounts.
3. Sign in as the bootstrap bank administrator and initialize the token. Register or verify the founder and participant institution organizations. Create the bank accounts and an institution administrator account for each participating institution, selecting or creating each user's wallet and choosing the appropriate application group and limits.
4. Complete deployment and setup for each participating organization. The institution administrator signs in and creates the remaining manager, officer, user, and auditor accounts for the organization.
5. After the institution manager accounts are created, as the bank administrator use the **Account Actions > Sync Roles** command on each manager account to add its Escrow ledger role. The institution administrator can assign the Manager application group but cannot grant the required Besu Notary role, which is displayed in the UI as Escrow. Verify all accounts, wallets, groups, roles, and limits.
6. Sign in as each bank and institution persona to verify that the expected workspace opens, the wallet and account context is correct, and that restricted actions are unavailable.

7. Run operations in the token life cycle: request and approve creation, transfer to the issuer, distribute or return value, approve participant transfers, and retire value.
8. Use the **Transactions** page and auditor workspaces to confirm balances, pending requests, operation IDs, and final ledger history.

Install the Deposit Token Application

Before you can use the application, you must configure the rich history database, install the application, and run an ORDS script.

The deposit token application uses Oracle AI Database to store indexed ledger history, with application-specific views so that the data can be queried. Oracle REST Data Services (ORDS) expose the views via authenticated REST routes. You must configure a rich history database profile for Oracle AI Database in Blockchain Platform Manager and then select that profile when you create an instance. For deployments with multiple organizations, each participating organization must have rich history enabled. Then, after the deposit token and account contracts are deployed, you run the bundled ORDS script against the same rich history database to create the required database views, ORDS REST routes, OAuth client, and optional analytics views.

Configure the Rich History Database Profile for the Deposit Token Application

You use Blockchain Platform Manager to create a profile for the rich history database.

You must meet the following prerequisites:

- Blockchain Platform Manager access to create rich history database profiles and create instances.
- An Oracle AI Database that is reachable from the Blockchain Platform Manager and from the computer where you run the ORDS script. The database must be provisioned.
- A database user account approved by the DBA to connect, read the rich history tables, create or replace views, and administer the ORDS objects that are required by the script. The DBA must grant the required object and ORDS administrative privileges for the selected schema.
- The following information from the DBA: the database endpoint and connection string or TNS service alias. The database username and password. Network approval for Blockchain Platform Manager to reach the database.
- If the database uses a wallet-based TLS connection, an extracted Oracle wallet .zip file, supplied by the DBA.

Do not place the wallet or any other production secrets in source control. Store database passwords, wallet files, OAuth client secrets, and bearer tokens in an approved secret management system.

1. In Blockchain Platform Manager, select **Configuration**.
2. Select **Rich History Database**.
3. Select **Add New**.
4. For **Database Type**, select **Oracle Database**.
5. Enter a unique name for **Config Name**. Use a name that makes the environment and purpose obvious, because this name is required later when you create an instance. For example, use a name of the form <environment>-deposit-token-rich-history-oracle.

6. Enter the database username, password, endpoint, and the connection string supplied by the DBA.
7. If the database connection requires a wallet, upload the wallet package.
8. Do not select **Use Database Blockchain Table** unless specifically required by your DBA.
9. Select **Save**.
10. Validate the profile by confirming that it is displayed when you select **Configuration > Rich History Database** and that the selected type is Oracle Database.

If Blockchain Platform Manager reports a connection failure, correct the database endpoint, connection string, credential, wallet, or network rule before attempting to create an instance.

You can now create the instance and deploy the application.

Create the Instance for the Deposit Token Application

After you configure the rich history database profile, you create the instance for the application.

Do not complete the following steps until you have configured the rich history database profile.

1. In Blockchain Platform Manager, Select **Instances**, then select **Create Instance**.
2. Enter the required settings, including archive node count, member node count, and smart contract governance mode.
3. For **Rich History DB Profile**, select the profile that you created, which must be an **Oracle Database** profile, not an **Internal DB** profile.

For more information about creating an instance, see *Create an Instance in Administering Oracle Blockchain Platform Enterprise Edition for Besu*.

4. To verify that rich history data is available, after the instance is created, deploy and initialize the deposit token application. For more information, see *One-Click Application Deployment*.
5. Run representative transactions, such as account creation, so that indexed rich history data is generated.

The ORDS Script uses the exact table prefix before the following rich history tables.

```
<BESU_TABLE_PREFIX>_more  
<BESU_TABLE_PREFIX>_store_state  
<BESU_TABLE_PREFIX>_store_hist
```

For example, if the tables are named `obpBesu_centralbank_123_more`, `obpBesu_centralbank_123_store_state`, and `obpBesu_centralbank_123_store_hist`, set the following prefix. Use the exact database object prefix, including case. Do not supply a view name, a table suffix such as `_more`, or a fabricated deployment label.

```
BESU_TABLE_PREFIX="obpBesu_centralbank_123"
```

You can now prepare and run the ORDS script.

Run the ORDS Script for the Deposit Token Application

After you create the instance and deploy the application, you can run the ORDS script.

You must meet the following prerequisites:

- Node.js version 20.19 or later, 22.13 or later, or 23.5 or later.
- You must have the ORDS script .zip file, which is downloadable from the service console.

Do not place production secrets in source control. Store database passwords, wallet files, OAuth client secrets, and bearer tokens in an approved secret management system. The script's committed .env file is a placeholder template only.

1. Extract the ORDS script .zip file to a working directory.
2. In the ORDSscript directory, open a terminal.
3. Run the following commands to confirm that the version of Node.js on your system is supported.

```
node --version
npm install
```

4. Configure the .env file by replacing placeholder values in the provided template. Do not add database passwords to this file; the command-line interface prompts for them interactively.

```
# Common ORDS configuration
CONNECTION_STRING="<Oracle connection string or TNS service alias>"
VIEW_PREFIX="<unique deployment namespace>"
MODULE_NAME="<unique ORDS module name>"
BASE_PATH="<unique/ords/base/path/>"
ITEMS_PER_PAGE="100"
ORDS_REST_BASE_URL="https://<ords-host>"
ALIAS_NAME="<REST-enabled schema alias>"
ROLE_NAME="<unique ORDS role>"
PRIVILEGE_NAME="<unique ORDS privilege>"
LABEL="<privilege label>"
DESCRIPTION="<deployment description>"
CLIENT_NAME="<OAuth client name>"
OWNER="<OAuth client owner>"
SUPPORT_EMAIL="<support email>"

# Besu Deposit Token configuration
BESU_TABLE_PREFIX="<exact prefix before _more, _store_state and _store_hist>"
BESU_TOKEN_CONTRACT="0x<deployed Deposit Token token proxy address>"
BESU_ACCOUNT_CONTRACT="0x<deployed Account proxy address>"
```

CONNECTION_STRING

If the CONNECTION_STRING parameter is a TNS alias, extract the wallet and configure the Oracle client environment so that the tnsnames.ora file can be found. The compressed wallet .zip file alone is not sufficient.

VIEW_PREFIX

Unique namespace for operational and optional analytics views. Use a different prefix for each deployment that shares an Oracle schema.

MODULE_NAME and BASE_PATH

These parameters define the ORDS module identity and URL path. Keep both deployment-specific and unique.

ALIAS_NAME

REST-enabled Oracle schema alias used in the public ORDS URL and token URL.

BESU_TABLE_PREFIX

Exact rich history table prefix; it determines the source `_more`, `_store_state`, and `_store_hist` tables.

BESU_TOKEN_CONTRACT

Deployed deposit token proxy contract address.

BESU_ACCOUNT_CONTRACT

Deployed account proxy contract address.

5. Enter the following command to run the script.

```
npm run start
```

- a. Enter the Oracle database username and password.
 - b. Select **Besu** as the blockchain platform.
 - c. Select **Deposit Token** as the application type.
 - d. If a prior run was incomplete, review the rollback prompt before choosing to revert objects.
 - e. Select **Yes** to create ORDS REST endpoints.
 - f. Select analytics views only when the deployment requires analytics output.
 - g. Select **Yes** to generate an OAuth client when a new client is required for this deployment.
6. If you selected endpoint and OAuth creation, retain the generated output in an approved secret store. The output contains the callable endpoint map, OAuth client ID, OAuth client secret, and a short-lived bearer-token response. Do not copy the client secret or bearer token to any source control system, file, wiki, or project management software that does not support approved secret handling. The following text shows sample output.

```
{
  "ORDEndpoint": "<generated endpoint URL or endpoint map>",
  "clientId": "<OAuth client ID>",
  "clientSecret": "<OAuth client secret>",
  "bearerToken": {
    "access_token": "<short-lived access token>",
    "token_type": "bearer",
    "expires_in": 3600
  }
}
```

7. If you selected OAuth client generation, retain the reported client ID and client secret in an approved secret store and use them to call an endpoint to verify system function. The following text shows the form of a token request

```
curl --request POST \
  "https://<ords-host>/ords/<ALIAS_NAME>/oauth/token" \
  --user "<client-id>:<client-secret>" \
  --header "Content-Type: application/x-www-form-urlencoded" \
  --data "grant_type=client_credentials"
```

Endpoint URLs use the following format.

```
https://<ords-host>/ords/<ALIAS_NAME>/<BASE_PATH>/<route>
```

The following command shows an example verification test.

```
curl --location \
  "https://<ords-host>/ords/<ALIAS_NAME>/<BASE_PATH>/
getAllDepositTokenAccounts?token_id=<TOKEN_ID>" \
  --header "Authorization: Bearer <access-token>" \
  --header "Accept: application/json"
```

8. Verify that the script ran successfully by running the following checks.

Blockchain Platform Manager profile

The Oracle AI Database rich history profile exists and is selected for the instance.

Rich history tables

The configured BESU_TABLE_PREFIX resolves to `_more`, `_store_state`, and `_store_hist` tables.

Indexed data

At least one representative deposit token transaction is shown in the rich history data.

Views

All operational views exist under the configured VIEW_PREFIX.

ORDS module

The configured module is published at the expected BASE_PATH.

OAuth

A client credentials request returns an access token.

API

A valid authenticated request returns HTTP 200. An empty `items` list is acceptable when no matching ledger data exists.

Isolation

Views, module, role, privilege, client, and alias names are unique for deployments sharing one Oracle database.

The following SQL queries are useful for validation.

```
SELECT view_name
FROM user_views
WHERE view_name LIKE UPPER('<VIEW_PREFIX>') || '_%'
```

```
ORDER BY view_name;

SELECT name, uri_prefix, items_per_page, status
FROM user_ords_modules
WHERE name = '<MODULE_NAME>';
```

Deposit Token Deployment Reference

The following information includes descriptions of the database objects that are created by the ORDS script, and troubleshooting tips.

The ORDS script creates the following operational views using the configured VIEW_PREFIX parameter.

```
<VIEW_PREFIX>_ACCOUNTS_V
<VIEW_PREFIX>_TRANSFER_EVENTS_V
<VIEW_PREFIX>_HOLDS_V
<VIEW_PREFIX>_ACTIVE_ROLES_V
<VIEW_PREFIX>_ACCOUNT_TX_HISTORY_V
<VIEW_PREFIX>_ACCOUNT_HISTORY_V
```

If you select analytics, the following analytics views are created, scoped to the deployment.

```
<VIEW_PREFIX>_ACCOUNTS_MOD
<VIEW_PREFIX>_TRANSACTION_MOD
<VIEW_PREFIX>_ACCOUNTS_TRANSACTION_MOD
```

The script also creates or configures the ORDS module, routes, schema alias, role, privilege, OAuth client, and client-role association using the configured values.

View Catalog and Query Logic

The following views are ordinary Oracle views in the selected schema; they do not modify the extractor tables.

View	Primary Source	Purpose/Logic
<VIEW_PREFIX>_ACCOUNTS_V	_more, _store_state	Builds the current account directory from createAccount calls and account/token state. It resolves account, user and organization identity, balances, hold balances, status, daily limits, token metadata, and creation transaction metadata.
<VIEW_PREFIX>_TRANSFER_EVENTS_V	_more	Normalizes token receipt logs into transfer-event rows. It extracts sender, receiver, amount, function, block/transaction metadata, and optional ~OBPRPCTX~.value.category context.

View	Primary Source	Purpose/Logic
<VIEW_PREFIX>_HOLDS_V	_store_hist, _more, _store_state	Reconstructs mint, burn, and transfer-hold records. It extracts operation/holding ID, accounts, notary, amount, expiry, token, category, description, and the latest lifecycle status. It also provides fallback rows when a current state row is not resolvable.
<VIEW_PREFIX>_ACTIVE_ROLES_V	_more	Applies role add/remove history and returns the latest active role assignment per account and role.
<VIEW_PREFIX>_ACCOUNT_TX_HISTORY_V	The preceding operational views	Produces account-facing transaction-history rows, including both sides of transfers and normalized mint/burn/hold lifecycle outcomes. It enriches from/to identities from the _ACCOUNTS_V view.
<VIEW_PREFIX>_ACCOUNT_HISTORY_V	_more, _ACCOUNTS_V	Produces account configuration history from account contract mutations, including daily limit revisions, timestamps, block numbers, and transaction IDs.
<VIEW_PREFIX>_ACCOUNTS_MOD (optional analytics)	_ACCOUNTS_V	Exposes account metadata in the analytics-compatible account shape.
<VIEW_PREFIX>_TRANSACTION_MOD (optional analytics)	_TRANSFER_EVENTS_V, _HOLDS_V, _store_hist	Exposes normalized business transactions, including transfer, mint, burn, request, hold, release, and resulting balances where available.
<VIEW_PREFIX>_ACCOUNTS_TRANSACTION_MOD (optional analytics)	_TRANSACTION_MOD, _ACCOUNTS_V	Adds source and destination organization context to analytics transaction records.

ORDS Routes

All routes are GET routes under <BASE_PATH>/ and require an OAuth bearer token. ORDS returns a collection envelope; an empty items array with HTTP 200 means that the request was valid but had no matching indexed records.

Route	Use and Response Behavior	Required Inputs	Optional Inputs and Default Behavior
getDepositTokenTransactionById	Returns one normalized ledger-transaction detail record.	token_id identifies the token; transaction_id identifies the Besu transaction.	None. No match returns an empty collection.

Route	Use and Response Behavior	Required Inputs	Optional Inputs and Default Behavior
getDepositTokenAccountTransactionHistory	Returns history for the account resolved by business identity.	token_id, org_id, and user_id identify the account.	startTime and endTime restrict the time range; omit either for no boundary. bookmark defaults to 0. pageSize uses the handler default.
getAllDepositTokenAccountTransactionHistory	Returns token-wide account transaction history.	token_id identifies the token.	startTime, endTime, bookmark, and pageSize behave as for the account-specific history route.
getDepositTokenAccountStatusHistory	Returns indexed status-history rows for a business account.	token_id, org_id, and user_id identify the account.	None.
getDepositTokenApproverActionHistory	Returns manager/approver decisions and final transfer-hold actions.	token_id identifies the token.	caller_account_id is the actor wallet; omit for all actors or provide it to scope to one wallet.
getDepositTokenHistory	Returns token-level history and supply-related information.	token_id identifies the token.	None.
getDepositTokenAccountsByRole	Returns accounts associated with a role.	role is the role name to match.	None.
getDepositTokenUsersByRole	Returns user/account role assignments for a role.	role is the role name to match.	None.
getDepositTokenOrganizationAccountsByRole	Returns organization accounts that have a role	org_id identifies the organization; role is the role name.	None.
getAllDepositTokenAccounts	Returns all indexed accounts for one token.	token_id identifies the token.	None.
getAllActiveDepositTokenAccounts	Returns accounts whose current indexed status is active.	token_id identifies the token.	None.
getAllSuspendedDepositTokenAccounts	Returns accounts whose current indexed status is suspended.	token_id identifies the token.	None.
getDepositTokenOrganizationAccounts	Returns all indexed accounts in one organization.	org_id identifies the organization.	None.
getPendingDepositTokenIssuance	Returns current pending transfer-hold operations only.	token_id identifies the token.	caller_account_id is the notary/manager wallet. Omit for the full queue; provide to scope to that wallet. Terminal operations are excluded.
getPendingDepositTokenRequest	Returns pending mint/burn/transfer requests only.	token_id identifies the token; request_type selects the supported request type.	None. Only operations whose latest lifecycle event remains pending are returned.

Route	Use and Response Behavior	Required Inputs	Optional Inputs and Default Behavior
getPendingRequest	Compatibility pending-request route.	token_id and request_type have the same meaning as getPendingDepositTokenRequest	None.
getDepositTokenAccountsByUser	Returns account records for one organization/user/token combination.	token_id, org_id, and user_id identify the account owner and token.	None.
getDepositTokenAccountHistory	Returns account configuration/history revisions.	account_address is the blockchain account/wallet address.	offset is a zero-based offset; limit caps rows. Omit either for the handler default.
getDepositTokenAccountStatus	Returns the current indexed account status.	token_id, org_id, and user_id identify the account.	None.
getDepositTokenRolesByAccount	Returns roles for a resolved account.	token_id is always required. Supply either account_address or org_id and user_id.	If account_address is supplied it is the direct lookup; otherwise organization/user lookup is used.
getAllDepositTokenOrganizations	Returns organizations represented by indexed accounts for a token.	token_id identifies the token.	None.
getDepositTokenMetadata	Returns indexed immutable token metadata.	token_id identifies the token.	None.
getDepositTokenRetiredQuantity	Returns retired/burned quantity information for a token.	token_id identifies the token.	None.
getDepositTokenAccountPolicies	Returns current account-policy records.	None.	account_policy_id selects one policy; account_address filters by account. Omit both to list all. bookmark defaults to 0; pageSize defaults to 20.
getDepositTokenAccountPolicyHistory	Returns account-policy revisions.	None.	account_policy_id and account_address filter revisions; omit both to list all. bookmark defaults to 0; pageSize defaults to 20.
getDepositTokenApprovalPolicies	Returns current approval-policy records.	None.	approval_policy_id selects one policy; omit it to list all. bookmark defaults to 0; pageSize defaults to 20.
getDepositTokenApprovalPolicyHistory	Returns approval-policy revisions.	None.	approval_policy_id filters revisions; omit it to list all. bookmark defaults to 0; pageSize defaults to 20.

Route	Use and Response Behavior	Required Inputs	Optional Inputs and Default Behavior
getPendingDepositTokenApprovals	Returns current operations awaiting approval.	None.	token_id and operation_id are optional filters. Omit operation_id to list all current pending operations. bookmark defaults to 0; pageSize defaults to 20.
getDepositTokenApprovalsByApprover	Returns approval/rejection decisions by an approver.	approver_address is the approver wallet address.	operation_id narrows results to one operation; omit for all decisions by that approver. bookmark defaults to 0; pageSize defaults to 20.
getDepositTokenApprovalsByOperationId	Returns approval/rejection events for one operation.	operation_id identifies the approval operation.	bookmark defaults to 0; pageSize defaults to 20.
getDepositTokenApprovalStatusByOperationId	Returns stored workflow status plus current active-policy resolution.	token_id and operation_id identify the workflow/hold.	None. It does not generate synthetic BLOCKED or AWAITING_MANAGER_COMPLETION statuses.

Pagination and timestamps: The bookmark parameter is a zero-based row offset and the pageSize parameter is case-sensitive camel case. For history routes, startTime and endTime use ISO-8601 timestamp text in the format YYYY-MM-DDTHH:MI:SS+/-HH:MI.

Endpoint inventory: The exact route inventory belongs to the script version packaged for the deployment. After the run, use the ORDS OpenAPI catalog or the generated endpoint output as the authoritative list for the client application.

Troubleshooting

Message or Condition	Resolution
NJS-516: no configuration directory set	A TNS alias was used without an accessible wallet/configuration directory. Extract the wallet and configure the Oracle client so that the tnsnames.ora file is available.
NJS-511 or ORA-12506	Listener, endpoint, service name, network allowlist, or wallet configuration issue. Validate the Oracle connection outside the script with the DBA.
OAuth token request returns 404	Verify the exact alias in /ords/<ALIAS_NAME>/oauth/token. The alias is not necessarily the Oracle username.
Endpoint returns 401	Obtain a fresh token and verify the OAuth client has the role/privilege associated with the configured module base path.
Endpoint returns 404	Verify host, alias, base path, route name, and that the module is published.
ORDS-25001	The endpoint reached ORDS but its SQL failed. Verify the table prefix, contract addresses, source data shape, and required request parameters.

Message or Condition	Resolution
ORA-20049: Cannot alter the URL mapping while the schema is enabled	The chosen ALIAS_NAME conflicts with an enabled schema mapping. Reuse the schema's existing alias, or have the ORDS administrator complete the approved disable/remap/enable procedure before running the script again.
Views contain no rows	Confirm the correct rich history profile was selected during instance creation, the table prefix is exact, contract addresses match the deployed proxies, and transactions have been indexed.
Objects conflict with another deployment	Use a distinct VIEW_PREFIX, module name, base path, role, privilege, OAuth client name, and alias.

One-Click Deploy the Deposit Token Application

One-click deployment prepares the required back end services and supplies the application configuration.

One-click deployment provisions and connects the following application components.

- Besu deposit token smart contract and deployment metadata, which provides the token operations and contract details that are used by the application.
- Besu wrapper API, which provides the application API for contract and account operations.
- Rich history database objects and scripts, which store transaction and operation history.
- Deposit token ORDS deployment and configuration, which exposes history and related back end services.
- Deposit token web application and service configuration, which connects the UI to the deployed back end services.

One-click deployment also prepopulates the **Setup** page with service endpoints, contract addresses, UWaaS and wallet settings, OIDC or LDAP settings, ORDS settings, and the packaged default customizations. Users do not need to enter these back end or identity provider values again. Only the deposit token ID is required on the **Setup** page. Optional user-facing customizations can also be changed there.

Before you begin, you need the following information.

- The target Oracle Blockchain Platform instance where the deposit token application will be deployed.
- The deposit token ID, to connect the application to the deployed token contract.
- Bootstrap user details that were supplied during when the smart contract was initialized, which identify the first bank administrator and provide initial administrative access.
- Optionally, display name, logo, currency, organization, or role label changes, which control the interface that users see after setup.

Complete the following steps for one-click deployment

1. Start the deposit token one-click deployment in the Oracle Blockchain Platform instance.
2. Wait until the smart contract, wrapper API, rich history, ORDS, and application components report a successful or ready state. This confirms that the application dependencies are available.

3. Open the application URL. The /set up page is displayed because deployment has completed the back end setup but the application configuration is not complete yet.
4. Enter the deposit token ID and review the prefilled values. Do not replace back end values unless you are making an actual change to the deployment.
5. Optionally, change the user-facing customization values, and then select **Apply Changes**. This validates the required fields, saves the token ID and customization values, combines them with the prefilled deployment configuration, and applies the resulting configuration to the application.

After you select **Apply Changes**, wait for the configuration update or redeployment to complete. The application then returns to the sign-in page. The bank administrator and financial institution administrator can sign in without a deposit token account if their selected wallet already has the matching Token Admin or Org Admin Besu contract role. Every persona still needs an eligible wallet; other personas also need an active deposit token account. One-click deployment does not create users, wallets, or deposit token accounts.

Configure the Deposit Token Application

You configure the application by checking application and connection settings, configuring the sign-in process, and configuring user interface settings.

The **Setup** page acts differently depending on how the deposit token application was deployed. With one-click deployment, the installation configuration from Oracle Blockchain Platform Enterprise Edition is used, including the identity provider configured for the instance, and this configuration is used to prefill the application values. With a local deployment, the deployer must deploy or connect to the required services and enter the application configuration manually.

Application and Connection Settings

Review or enter the following settings.

The network selection is fixed by the deployed runtime environment. Do not change prefilled contract addresses, service identifiers, OIDC settings, or ORDS settings unless the replacement services belong to the same deployment and the change is part of an approved deployment procedure.

Setting	One-Click Deployment	Local Deployment
Deposit token ID	Enter the token ID deployed for this application.	Enter the token ID manually.
Protocol and authentication mode	Prefilled from the Oracle Blockchain Platform deployment; normally review only.	Enter the approved local protocol and authentication mode.
Besu wrapper API and access token URL	Prefilled by Blockchain Platform Manager.	Enter the URLs for the locally deployed services.
Oracle Blockchain Platform Instance client ID and client secret	Prefilled from the Oracle Blockchain Platform instance configuration; the secret remains masked.	Enter the local client values when the deployment requires them.
Account and deposit token contract addresses	Prefilled from the deployed Besu contracts.	Enter the addresses returned by the local contract deployment.
UWaaS wallet URL, organization ID, and wallet service instance ID	Prefilled from the Oracle Blockchain Platform instance setup.	Enter the local UWaaS and wallet service values.

Setting	One-Click Deployment	Local Deployment
Default caller address	Prefilled when required by the deployment.	Enter it when the local deployment requires a default caller.
ORDS OAuth URL, client ID, and client secret	Prefilled from the ORDS deployment managed by Blockchain Platform Manager; the secret remains masked.	Enter the values for the locally deployed ORDS services.

Sign-In Process

For one-click deployment, the identity provider is the same one configured for the Oracle Blockchain Platform installation. OIDC or LDAP values are prefilled by Blockchain Platform Manager. Review the values, but do not replace these values unless the Oracle Blockchain Platform deployment configuration is intentionally changed.

For a local deployment, no identity provider configuration is inherited automatically. Enter the issuer URL, client ID, client secret if required, scopes, user-ID claim, exact redirect URI, and post-logout redirect URI. Register the same URLs in the selected identity provider.

OIDC authenticates the identity. The application still checks the selected organization, wallet service instance, active deposit token account, application group, and smart contract role before allowing an operation. Configure multi-factor authentication in the identity provider.

Application Presentation

Open the customization cards and review the following presentation settings.

Customization Area	Controls
Token and deposit token account display names	Names shown in headers, account cards, balances, and transaction forms.
Currency and number format	Currency symbol, decimal display, grouping, and symbol position. For U.S. dollars, select USD — US Dollar (\$) and confirm the number preview. Besu token decimals come from the deployed contract and cannot be changed during configuration. If the displayed precision is wrong, correct the contract deployment instead of changing display settings to hide the mismatch. The token's contract-defined decimal precision is read-only in Setup and is not the same as the currency symbol and number formatting used for display.
Login branding	Message and image shown before sign-in.
Application background	Background color used across the application shell.
Transaction labels	User-facing names for transaction types.
Organization display names, logos, and order	Presentation of organizations in dropdowns and workspace navigation. Live UWaaS IDs, names, sequences, and founder tags remain authoritative.
Application group labels	Names shown for bank and institution personas.
Ledger role labels	Names shown for smart-contract permissions such as Token Admin, Minter, Burner, and Escrow/Notary.

Apply the Configuration

For one-click deployment, the **Apply Changes** command validates required fields, saves the token ID and optional customizations, combines them with the configuration from Blockchain Platform Manager, and applies the resulting configuration to the application. When the configuration update or redeployment is complete, the application returns to the sign-in page.

For a local deployment, the **Apply Changes** command validates required fields and writes the local configuration to the `.env.local` file. Masked secret fields can be left blank to retain their current values. You must restart the application when requested. The application then returns to the sign-in page.

Deposit Token Sample Application Workflow

After you install and configure the sample deposit token application, you can use it in scenarios where a system owner (a bank) and participant organizations (other institutions) interact in an interbank market.

The sample application supports twelve roles, or personas. Each role has a different interface and set of operations that support the entire workflow of token management in the deposit token scenario.

Bank Roles

- Bank administrator: Configures tokens and organizations; creates wallets and deposit token accounts; sets limits, purposes, groups, and roles. Maintains compliance controls and approval policies.
- Bank approver: Completes assigned amount-based or sequential transfer approvals. The configured bank approver or sequential bank approver chain completes the approval policy decision steps for a policy-controlled transfer.
- Bank creator: Requests token creation and transfers the created value to a bank issuer.
- Bank issuer: Issues value to participants and returns value for retirement.
- Bank manager: Reviews assigned creation, issuance, pending transfer, and retirement requests. A bank manager approves or rejects requests to create or retire tokens. For issuance transfers that are subject to an approval policy, the manager can complete the transfer only after the required approver sequence is satisfied; the manager can reject a transfer while it is pending.
- Bank retiree: Submits retirement or burn requests.
- Bank auditor: Reviews network activity, balances, pending requests, and history, without changing data.

Institution Roles

- Institution administrator: Manages participant wallets, accounts, application groups, status, and limits.
- Institution manager: Approves or rejects transfers for the participant organization. An institution manager reviews transfers associated with its organization's escrow account. The manager can approve a direct manager action, complete a policy-controlled transfer only after the required approver sequence is satisfied, or reject a pending transfer.
- Institution officer: Deposits value to the bank and transfers value to permitted users.
- Institution user: Transfers value to permitted recipients.

- Institution auditor: Reviews participant/organization activity, without changing data.

Onboarding

After you install and configure the application, you prepare users, wallets, deposit token accounts, and ledger roles. Complete the following steps before you run token operations.

1. The bank administrator logs in first. Select the bootstrap wallet that was granted the Token Admin Besu role when the smart contract was initialized. This is the initial administrative entry point even before a deposit token account exists.
2. The bank administrator verifies the organization registry. Register or review the founder and participant organizations, confirm the founder sequence (1), and verify the organization order and display values.
3. The bank administrator initializes the token and creates the initial accounts. Create wallets and deposit token accounts for bank personas and for the institution administrators who will administer participating organizations. Select the correct wallet, application group, and account limits.
4. Each participating organization is deployed and configured. Use the same one-click or local deployment process for the participant application. One-click deployment uses the Oracle Blockchain Platform configuration; local deployment requires manual configuration.
5. The institution administrator logs in for the participant organization. Load or create the organization's wallets, then create manager, officer, user, and auditor accounts using the permitted application groups and limits.
6. The bank administrator configures or reviews governance and user compliance controls and creates or activates amount-based approval policies. A policy without approvers can settle a matching transfer immediately; a policy with approvers creates the configured approval sequence.
7. The bank administrator synchronizes manager roles. After each institution manager account is created, use the **Account Actions > Sync Roles** command on that account to apply its Escrow ledger role. Application groups control workspace access; ledger roles control contract operations.
8. Each persona logs in and verifies access. Select the intended wallet when more than one is available, confirm the correct workspace opens, and verify that permitted and restricted actions match the assigned application group and ledger role. Use the persona sequence in the following table.
9. After all persona configuration and access checks are complete, follow the application workflow to work with the deposit token life cycle.

Persona Sequence

Sequence	Persona	Responsibility	Information
1	Bank administrator	Completes bank, organization, wallet, account, and role setup.	Bank Administrator
2	Institution administrator	Sets up participant wallets and accounts.	Institution Administrator
3	Bank creator	Submit requests to create tokens.	Bank Creator
4	Bank manager	Approves or rejects creation requests.	Bank Manager

Sequence	Persona	Responsibility	Information
5	Bank approver	Completes assigned approvals for configured amount-based or sequential transfers.	Bank Approver
6	Bank issuer	Issues value to participants and returns value for retirement.	Bank Issuer
7	Institution manager	Approves direct manager actions, completes policy-controlled transfers only after the required bank approver sequence, or rejects pending transfers.	Institution Manager
8	Institution officer	Deposits and transfers value.	Institution Officer
9	Institution user	Transfers value to permitted recipients.	Institution User
10	Bank retiree	Submits retirement requests after value is returned.	Bank Retiree
11	Bank auditor	Reviews bank activity and history.	Bank Auditor
12	Institution auditor	Review participant organization activity and history.	Institution Auditor

Application Workflow (Token Life Cycle)

After deployment and setup, the personas in the scenario engage in the following tasks as part of the complete token life cycle. Note that transfer approval is conditional, not a fixed lifecycle stage: after a policy-controlled transfer is submitted, the configured bank approver completes the current approval step before that transfer settles.

1. The bank creator requests token creation and moves approved value. Open the Create Tokens page, enter the quantity and optional memo, review the generated operation ID, and submit the request. After the bank manager approves the request, submit the transfer of approved value to the bank issuer. The applicable approval policy determines whether the transfer completes immediately or is routed for approval.
2. The bank manager approves or rejects bank requests. Open the **Creation** queue to approve or reject token creation. Approval credits the creator; rejection leaves the request uncredited. The same manager later approves or rejects burn requests from the bank retiree and can complete an eligible transfer only after its configured approval sequence is satisfied.
3. The bank issuer distributes or returns value. Transfer value to an institution officer, or use the retirement path to transfer value to bank retiree. The active amount-based policy determines whether an eligible transfer settles immediately or waits for the configured approval sequence.
4. The bank approver completes assigned approval steps. When an amount-based policy routes a submitted transfer to the bank approver: Open the Bank Approvals page, review the transaction and current sequence position, and then approve to advance or complete it or reject it to stop the transfer. Multiple approvers act in sequence.

5. The institution manager reviews manager/escrow transfers. Review pending participant transfers that are associated with the manager/escrow account. Approve a direct manager action or complete an eligible transfer after its configured approval sequence is satisfied; reject when the transfer must not settle.
6. The institution officer moves participant value. Transfer value to bank issuer, another institution officer, or an institution user. The active policy determines whether the transfer settles immediately or is routed to the configured bank approver sequence.
7. A institution user transfers permitted value. Transfer value to an allowed institution officer or institution user. The same policy rules apply: no configured approver can settle immediately. Multiple approvers act in sequence.
8. The bank retiree requests retirement. After receiving value, open the Retire Tokens window, enter the amount and required details, and submit the burn request. The bank manager approves or rejects the request. Approval removes the value from circulation; rejection leaves the burn incomplete.
9. The bank auditor reviews bank activity. Use the auditor workspace, filters, transaction rows, detail dialogs, operation IDs, balances, approval outcomes, and history to review the bank activity that is available in read-only format.
10. The institution auditor reviews participant activity. Use the participant auditor workspace to review the organization's accounts, transfers, approvals, balances, operation IDs, and history. The workspace of the institution auditor workspace is read-only and is limited to their own organization.

Approval policies: Token creation and retirement use the configured bank manager request queues. Issuer, officer, user, creator-to-user, and return transfers use the active amount-based policy. A request can settle immediately if no approver is configured. When an approval is required, the policy routes the request to the configured bank approver account; one approver must decide it, and multiple approvers act in sequence. A rejection stops the pending transfer; it does not settle.

After each request or decision, refresh the sender, recipient, manager, and auditor views as applicable. Confirm the final status and balance before submitting another operation.

Bank Administrator

The bank administrator has the first operational workspace after deployment.

The bank administrator initializes the token, registers organizations, creates wallets and deposit token accounts, maintains roles and limits, reviews activity, and configures the application presentation.

Log in as the bank administrator user with the wallet granted the Token Admin Besu contract role when the smart contract was initialized. The wallet must be eligible for the configured organization and wallet service instance. This user can log in before a deposit token account is created. Create the remaining users' wallets and deposit token accounts before they log in.

Before you begin the administrator tasks, the application must be deployed, either by one-click deployment or local deployment, and configured so that it is connected to the Besu contracts, wrapper API service, wallet service, and other supporting services. You must also have the following information.

- Bank administrator login details from smart contract initialization, to open the application and complete administrative setup before an operational deposit token account exists.

- The token ID and token description supplied for the deployment. The token description is handled during token initialization; account creation uses the selected token ID and account details.
- Organization names, unique organization IDs, and the intended founder/participant order. Organizations must be registered before accounts can be created for them.
- User IDs and eligible wallets for the users who will operate the application. Each operational account is linked to a user and an active wallet.
- The application group and daily limits required for each user. The selected group controls the workspace and the configured limits control account usage.

Typically the bank administrator completes tasks in the following order.

1. Open the system dashboard.
2. Initialize the token if deployment has not already initialized it.
3. Register the founder and participant organizations, and then confirm their display names and sequence order.
4. Create wallets and deposit token accounts for the bank administrator and other bank personas.
5. Create accounts for participating organization administrators. After participating organization administrators create their manager accounts, use the **Sync Roles** command on those accounts to add the escrow ledger role.
6. Review or update token details and configure the transaction purposes before operational token transfers begin.
7. Review the account list, organization cards, KPIs, and analytics dashboard.
8. Configure the application settings and customize the application presentation.

The following tasks are typical ones that a bank administrator completes.

Initialize Token

Initializing the token is the first task that the bank administrator must complete.

1. Select **Token Management > Initialize Token**. The Initialize Token window opens.
2. Confirm that the token ID is correct.
3. Edit the customer-facing token description as needed, including the **Currency**, **Currency Description**, and **Conversion Rate**. Optionally, select **Purpose** and then select one of the configured purposes.
4. Select **Initialize Token** to complete the process.

Register Organizations

You must register organizations before you create accounts for them. Use the organization sequence to control the order in which organizations are displayed in the application.

1. Select **Participants > Organizations**. The Manage Organizations window opens.
2. Enter an **Organization Name**, **Organization ID** (the unique ID used by the network registry), and a **Sequence** number. The founder organization is always first in the sequence. If the requested sequence is already in use, the selected organization and later entries move to preserve a valid order. The application also reconciles missing, duplicate, or non-consecutive values using the registry data and creation order. If the live registry contains more than one founder, correct the founder tags in the registry before continuing.

3. Select Register Organization.**Create Wallets and Deposit Token Accounts**

Complete the following steps to create accounts.

1. Select **Participants** > **Create Bank Account**. The Create Token Bank Account window opens.
2. Select the **Bank Name**, and enter the **Bank User ID**.
3. Select **Load wallets** for the configured wallet service instance.
4. Select the user's wallet, or if the user has no eligible wallet, create a wallet. To create a wallet, select **Create new wallet**. Loading or creating wallets for another organization might require that organization's wallet service instance ID.
5. Select the **Group** and set the account limits.
The following table describes the account information fields.

Field	Description
Token ID	The ID of the token that was initialized for the system.
Bank Name	A list of all registered organizations in the network.
Org ID	The organization ID of the bank, which is determined by the selected Bank Name .
Bank User ID	A user ID for the account.
Role	A list of supported roles. Role information is supplied by the selected application group. Roles are assigned by the application during account creation.
Group	A list of application groups (personas).
Daily Maximum Amount	The maximum amount of tokens that can be transacted per day.
Daily Maximum Transactions	The maximum number of transactions allowed per day.
Wallet	Select an active wallet with no existing token account. Use Load wallets to retrieve eligible wallets or Create new wallet when the deployment permits wallet creation.

6. Select **Create Account**. The account details are displayed.

The selected application group and its associated roles are assigned automatically, with the exception of a institution manager. After creating a participating institution manager account, open its account actions menu and select **Sync Roles** to apply the required escrow ledger role. Application groups determine the workspace shown in the UI; ledger roles determine contract operations.

Work with Governance

The bank administrator can access the Bank Governance workspace to maintain user compliance controls and amount-based transfer approval policies.

To adjust compliance controls for a specific user, select the user and update the following fields.

KYC

The identity verification state used by configured compliance checks.

AML

The anti-money-laundering state used by controlled transfer checks.

Risk

The risk value associated with the user for policy evaluation.

Restriction

The restriction state, which can prevent a controlled operation.

To adjust approval policies, select the policy and update the following fields.

Amount range

The lower and upper value amounts for the policy.

Status

Activates or deactivates the policy.

Approver organization

The organization responsible for an approval step.

Approver user ID

The user responsible for an approval step.

Order

Moves approvers into a sequence.

A policy with no approvers can settle a matching transfer immediately. With one approver, the transfer is routed to that approver, who must explicitly approve or reject it. A policy with multiple approvers advances sequentially. Completed approval steps remain recorded. For a request that is still pending, remaining approval routing is evaluated using the active policy. A policy's organization ID and user ID must match an active bank approver account with an eligible wallet.

Additional Administrator Tasks

Select **Token Management > Manage Purposes** to add, edit, or remove the purpose values that are available for transfers. Removing a purpose does not rewrite existing transaction history.

Select **Insights > Analytics Dashboard** to open the read-only dashboard. From the summary cards, select the **Accounts**, **Transactions**, or **Token** view. Choose an organization and status for a narrower set of results. Select **Refresh Analytics** when the page was opened before a recent operation completed. A "No analytics data" message means the selected view has no available records or is still loading; adjust the filters or refresh the page.

Select **Account > My Account** to open a read-only page for the currently logged-in user. You can use this page to confirm the account selected at sign-in, its organization and token, its status and balance, its limits, and the changes recorded for the account.

Troubleshooting

If organization or account data is unavailable, verify the UWaaS, wrapper API, smart contract, wallet, and ORDS configuration. If a role is missing, use the **Sync Roles** command and ask the user to log out and log in again so that the session is rebuilt.

Bank Approver

The bank approver completes transfer approvals as configured in the bank's governance policies.

This persona is distinct from the bank manager.

Before you can complete bank approver tasks, you must have an active deposit token account and an eligible wallet. You must be a member of the Bank Approver application group and must be assigned to the current approval position. Before an administrator adds a user to an approval policy sequence, they must confirm the exact organization ID and user ID that belong to this active bank approver account.

Typically the bank approver completes tasks in the following order.

1. Open the Requests window and locate a request assigned to the signed-in user and current approval position.
2. Review the request row and approval sequence information, and then select **Approve** or **Reject** and confirm the decision. **Approve** advances the request to the next approver or completes it when this is the final step. **Reject** stops the pending transfer; it does not settle.
3. Refresh the Requests and History windows, and then verify the new statuses and balances.

Bank Auditor

The bank auditor has a read-only workspace for reviewing account activity, transaction history, and audit policy conditions across the bank and participating organizations.

The bank auditor is a read-only account. The auditor cannot create or update accounts, move value, approve requests, synchronize roles, or change setup.

Before you can complete bank auditor tasks, you must have an eligible active deposit token account that is linked to your wallet. You must be a member of the Bank Auditor application group with its assigned ledger role. Finally, there must be transaction data stored in the configured rich history service.

Typically the bank auditor completes tasks in the following order.

1. Confirm the auditor identity, organization, token, and account status.
2. Review the organization and account status cards.
3. Enable any auditing policies that are required for review.
4. Select the relevant organization or account history control.
5. Apply transaction filters, refresh the history, and sort or paginate the results.
6. Use the transaction ID, timestamp, amount, counterparty, and outcome to reconcile the record.
7. Export the visible history when an offline review is required.

Audit Policies

Audit policies lets the auditor define review criteria for unusual or important activity. Audit policies do not change ledger data and do not approve or reject transactions. You must enable a policy before you can enter or edit the policy parameters.

Policy	Parameters
Single transfer amount	Choose a comparison such as >= , <= , or = , and then enter an amount.
Number of transfers exceeding	Enter a count, and then choose the time window: Day , An Hour , Weekly , or Monthly .
Daily transfer total amount exceeding	Enter the amount threshold.
Transfers to specific accounts	Select the accounts.
Transfers from specific accounts	Select the accounts.

Multiple complete, enabled audit policies work together to narrow the displayed history: a record must satisfy each applicable condition. These are UI review criteria, not persistent ledger rules.

Transaction Filters

After you load records from an organization or account history, you can use the following filters to find specific transactions.

Filter	Parameters
Transaction ID	Enter an operation or transaction identifier.
Amount	Select a comparison and enter an amount.
Balance	Select a comparison and enter a balance.
Transaction Type	Select a transaction type.
Initiator	Enter a user or organization value.
Account ID	Enter an account identifier.
Purpose/Memo	Enter the text to search for.
Start Date / End Date	Select the beginning and ending dates.

Selecting a transaction history row opens a read-only Transaction details window, which includes the following fields.

Section	Information Displayed
Summary	Transaction ID, Holding ID, Purpose/Memo, Block Number, Transaction Time, Transaction Type, Amount, and Token ID.
From	Organization, User ID, and From Account ID. The From party displays - for a creation transaction.
To	Organization, User ID, and To Account ID. The To party displays - for a retirement transaction.

Troubleshooting

If data is missing, select **Refresh History**, clear restrictive filters, verify the date range, and confirm the history service and transaction ID. If an unexpected organization or action is visible, verify the account's assigned application group and permissions.

Bank Creator

The bank creator requests new deposit token supply and transfers available value to eligible bank recipients.

The bank creator can request new supply and transfer available value to permitted recipients. The creator cannot approve their own creation requests, change account limits, assign roles, or change setup.

Before you can complete bank creator tasks, you must have an active deposit token account and an eligible wallet. You must be a member of the Bank Creator application group with the assigned Minter ledger role. There must also be a user who is an eligible manager approver with the Escrow role, and a user with an eligible recipient account for transfers.

Typically the bank creator completes tasks in the following order.

1. Confirm the creator account is active and review the available balance.
2. Select **Create Tokens**, enter the quantity, and optionally add a memo. Review the active escrow user shown by the form; if more than one eligible escrow user is available, select the intended user.
3. Submit the request and record its request or transaction ID.
4. Wait for the bank manager to approve or reject the request.
5. After approval, refresh the creator page and confirm that the balance and history are updated.
6. Select **Issue** beside the intended bank issuer, confirm the recipient user and escrow context, optionally enable and select a purpose, enter the quantity, and submit the transfer.
7. Refresh history and confirm the transfer status and transaction ID.

The following tasks are typical ones that a bank creator completes.

Request Token Creation

1. Select **Create Tokens**. The Create Deposit Token Request window is displayed.
2. Enter a **Quantity** and an optional **Memo**. If multiple eligible managers are available, select an **Approver User ID**.
3. Select **Request Mint**. The request is sent for manager approval.

Issue Tokens

1. Select **Issue** on an eligible bank issuer card. The Transfer window is displayed.
2. Verify the recipient organization for **Transfer Token To** and the eligible bank issuer for **To User ID**, and enter a **Quantity**. Optionally, select **Purpose** and then select one of the configured purposes.
3. If more than one **Approver User ID** (active escrow user) is available for the sending organization, select the intended approver.
4. Select **Transfer**. The applicable approval policy determines whether the transfer completes immediately or is routed for approval.

Transaction Filters

After you load records from an organization or account history, you can use the following filters to find specific transactions.

Filter	Parameters
Transaction ID	Enter an operation or transaction identifier.
Amount	Select a comparison and enter an amount.
Balance	Select a comparison and enter a balance.

Filter	Parameters
Transaction Type	Select a transaction type.
Initiator	Enter a user or organization value.
Account ID	Enter an account identifier.
Purpose/Memo	Enter the text to search for.
Start Date / End Date	Select the beginning and ending dates.

Selecting a transaction history row opens a read-only Transaction details window, which includes the following fields.

Section	Information Displayed
Summary	Transaction ID, Holding ID, Purpose/Memo, Block Number, Transaction Time, Transaction Type, Amount, and Token ID.
From	Organization, User ID, and From Account ID. The From party displays - for a creation transaction.
To	Organization, User ID, and To Account ID. The To party displays - for a retirement transaction.

Troubleshooting

If no escrow user is available, ask the bank administrator to verify active manager accounts and Escrow roles. If an amount is rejected, check the positive value, decimal precision, balance, and limits. If a request remains pending, give the request ID to the selected bank manager. If the account is inactive, ask the bank administrator to review its status.

Bank Issuer

The bank issuer sends deposit tokens to eligible participating organization officers and returns value to an eligible bank retiree account.

The bank issuer can initiate participant issuance and retirement transfers shown by the application. Issuers cannot approve their own participant issuance, create accounts, assign roles, change limits, or change setup.

Before you can complete bank issuer tasks, you must have an active issuer deposit token account with an available balance. You must be a member of the Bank Issuer application group. There must also be active officer accounts in participating organizations, an active bank retiree account, and an active escrow user for the sending organization..

Typically the bank issuer completes tasks in the following order.

1. Confirm the issuer account is active and review pending and available balances.
2. Expand the participant organization and select an active officer.
3. Select **Issue**, review the approver organization and approver user ID context, optionally enable and select a purpose, enter the quantity, and submit.
4. Record the operation ID and follow its status. If approval is required, the active policy routes it to the configured bank approver sequence.
5. Refresh history and confirm the issuance status and balance after approval or rejection.

6. For retirement, select an active retiree account, optionally enable and select a purpose, enter the quantity, and submit the **Retirement** transfer.
7. Refresh history and confirm the retirement result.

The following tasks are typical ones that a bank issuer completes.

Issue Value to an Institution Officer

1. Expand a participant organization, choose an active officer, and select **Issue**. The Transfer window is displayed.
2. Enter a **Quantity** within the issuer balance, recipient limits, and supported precision. Optionally, select **Purpose** and then select one of the configured purposes.
3. If more than one **Approver User ID** (active escrow user) is available for the sending organization, select the intended approver.
4. Select **Transfer**. The active policy determines whether the transfer settles immediately or is routed to the configured bank approver sequence.

Return Value for Retirement

1. Select **Retirement** for an active eligible retiree account. Retiree eligibility is based on account and role data, not the displayed name. The Transfer window is displayed.
2. Enter a **Quantity**. Optionally, select **Purpose** and then select one of the configured purposes.
3. Select **Transfer**. The transfer is evaluated by the active approval policy. It can settle immediately or remain pending for the configured bank approver sequence.

Transaction Filters

After you load records from an organization or account history, you can use the following filters to find specific transactions.

Filter	Parameters
Transaction ID	Enter an operation or transaction identifier.
Amount	Select a comparison and enter an amount.
Balance	Select a comparison and enter a balance.
Transaction Type	Select a transaction type.
Initiator	Enter a user or organization value.
Account ID	Enter an account identifier.
Purpose/Memo	Enter the text to search for.
Start Date / End Date	Select the beginning and ending dates.

Selecting a transaction history row opens a read-only Transaction details window, which includes the following fields.

Section	Information Displayed
Summary	Transaction ID, Holding ID, Purpose/Memo, Block Number, Transaction Time, Transaction Type, Amount, and Token ID.
From	Organization, User ID, and From Account ID. The From party displays - for a creation transaction.

Section	Information Displayed
To	Organization, User ID, and To Account ID. The To party displays - for a retirement transaction.

Troubleshooting

If no participant officers are displayed, ask the participant administrator to create and activate officer accounts. If no retiree is displayed, ask the bank administrator to verify the account and assigned role. If an amount fails validation, check the balance, daily limits, recipient status, and decimal precision. If an issuance remains pending, use the operation ID to review the current approval status.

Bank Manager

The bank manager reviews and approves or rejects pending creation, issuance, and retirement operations that are assigned to them.

The bank manager can approve or reject assigned pending operations. They cannot create accounts, change limits, assign roles, edit token configuration, or change setup.

The bank manager approves or rejects requests to create or retire tokens. For issuance transfers governed by an active approval policy, the configured bank approver or bank approver sequence performs the required policy approval steps. The manager does not perform or replace those approval policy steps. After the required approver sequence has been successfully completed, the manager can complete the eligible issuance transfer. The manager can also reject the transfer while it is pending.

Before you can complete bank manager tasks, you must have an active deposit token account and an eligible wallet. You must be a member of the Bank Manager application group and have the Escrow ledger role. There must be pending requests assigned to you or your organization.

Typically the bank manager completes tasks in the following order.

1. Review pending operations and the pending totals by organization.
2. Open the page that matches the request: **Issuance, Creation, or Retirement**.
3. Read the operation ID, parties, amount, purpose or memo, and request type.
4. Approve or reject a request, or choose multiple eligible requests and select **Approve Selected**. (You cannot reject multiple requests at a time.) For issuance transfers governed by an active approval policy, the configured bank approver or bank approver sequence performs the required policy approval steps. The manager does not perform or replace those approval policy steps. After the required approver sequence has been successfully completed, the manager can complete the eligible issuance transfer. The manager can also reject the transfer while it is pending.
5. Refresh data and confirm that the request leaves the **Pending** page.
6. Select **History** and verify the terminal status and affected amount.

Troubleshooting

If an action is disabled, verify that the manager account is active and that the row is still pending. If a request is removed, check the **History**, because it may already be in a terminal state. If an amount or party is missing, refresh the data and verify the source operation. If a request is assigned elsewhere, the bank administrator must verify the approver organization, user, and escrow role. For a policy-controlled transfer, the administrator must verify that the

policy's exact organization and user identify an active bank approver account with an eligible wallet.

Bank Retirer

The bank retirer receives value returned by the issuer and submits requests to remove that value from circulation.

The bank retirer can send burn requests and return available value to eligible issuers. They cannot approve their own retirement requests, create accounts, assign roles, change limits, or change setup.

Before you can complete bank retirer tasks, you must have an active retirer deposit token account and an eligible wallet. You must be a member of the Bank Retirer application group with the Burner ledger role. There must also be value (tokens) returned by a bank issuer, an eligible manager approver, and an active bank issuer account.

Typically the bank retirer completes tasks in the following order.

1. Confirm the retirer account is active and review the total retired and balance amounts.
2. If value has not arrived, ask the bank issuer to use **Transfer** to send value to this account.
3. Select **Retire Tokens**, choose the manager approver, enter the quantity, and optionally add a memo.
4. Submit the burn request and record the request ID.
5. Wait for the bank manager to approve or reject the request.
6. Refresh history and confirm the retirement status, balance, and total retired value.

The following tasks are typical ones that a bank retirer completes.

Retire Tokens

1. Select **Retire Tokens**. The Request Burn Tokens window is displayed.
2. Enter a **Quantity** and an optional **Memo**. If multiple eligible managers are available, select an **Approver User ID**.
3. Select **Request Burn**. The request is sent for manager approval.

Return Value to Issuer

1. Select **Transfer** for an eligible bank issuer.
2. Select the **To User ID**, enter a **Quantity**. Optionally, select **Purpose** and then select one of the configured purposes.
3. Select **Transfer**. The return transfer is evaluated by the active approval policy and may settle immediately or remain pending for the configured approval sequence. If approved, the value is returned to the selected issuer and the approver organization and approver user ID context used for the approval-aware transfer are displayed.

Transaction Filters

After you load records from an organization or account history, you can use the following filters to find specific transactions.

Filter	Parameters
Transaction ID	Enter an operation or transaction identifier.

Filter	Parameters
Amount	Select a comparison and enter an amount.
Balance	Select a comparison and enter a balance.
Transaction Type	Select a transaction type.
Initiator	Enter a user or organization value.
Account ID	Enter an account identifier.
Purpose/Memo	Enter the text to search for.
Start Date / End Date	Select the beginning and ending dates.

Selecting a transaction history row opens a read-only Transaction details window, which includes the following fields.

Section	Information Displayed
Summary	Transaction ID, Holding ID, Purpose/Memo, Block Number, Transaction Time, Transaction Type, Amount, and Token ID.
From	Organization, User ID, and From Account ID. The From party displays - for a creation transaction.
To	Organization, User ID, and To Account ID. The To party displays - for a retirement transaction.

Troubleshooting

If no returned value is available, ask the bank issuer to transfer it. If no approver is available, ask the bank administrator to verify active manager accounts and Escrow roles. If a quantity is rejected, check the balance, limits, and decimal precision. If the account is inactive, ask the bank administrator to review the account.

Institution Administrator

The institution administrator manages wallets and deposit token accounts for a participating organization..

A participant account created with the Org Admin application group adds the corresponding Org Admin role automatically. Institution administrators cannot grant minter, burner, token administrator, or bank escrow roles. The bank administrator must use the **Sync Roles** to apply the escrow role to participating organization manager accounts.

Before you can complete institution administrator tasks, you must be a member of the Institution Admin application group for the selected organization. A deposit token account is not required to log in as an institution administrator if the selected eligible wallet already has the Org Admin Besu contract role. The organization and wallet service instance must be registered before creating participant accounts.

Typically the institution administrator completes the following tasks.

- Loads or creates wallets
- Creates accounts for organization users, selecting the institution application group that matches the user's intended persona and responsibilities. Org Admin and Org Auditor roles are added for their corresponding groups. The bank administrator must use the **Sync Roles** command for the manager (Escrow) role.

- Edits limits and account states
- Activates or suspends accounts
- Inspects application groups and contacts the bank administrator when a ledger role must be synchronized.

The following tasks are typical ones that an institution administrator completes.

Create Wallets and Deposit Token Accounts

Complete the following steps to create accounts.

1. Select **Create Bank Account**. The Create Token Bank Account window opens.
2. Enter the **Bank User ID** and confirm the organization and token IDs.
3. Select **Load wallets** and choose the user's wallet, or select **Create new wallet**.
4. Select the **Group** and set the account limits.
The following table describes the account information fields.

Field	Description
Token ID	The ID of the token that selected for the organization.
Organization	The selected organization name.
Organization ID	The registry identifier of the bank.
Bank User ID	The user ID registered for the participant organization.
Role	A list of supported roles. Role information is supplied by the selected application group.
Group	A list of application groups (personas).
Daily Maximum Amount	The maximum amount of tokens that can be transacted per day.
Daily Maximum Transactions	The maximum number of transactions allowed per day.
Wallet	Select an active wallet with no existing token account. Use Load wallets after entering the user and organization to retrieve eligible wallets or select Create new wallet .

5. Select **Create Account**. The account details are displayed.

Troubleshooting

If there is a wallet mismatch, verify the user, organization, and wallet service instance. If a role is missing after account creation, request that the bank administrator run the **Sync Roles** command. If there is a duplicate or ambiguous account, verify the selected wallet and account address before retrying.

Institution Auditor

The institution auditor has a read-only workspace for reviewing account activity, transaction history, and audit policy conditions for a single institution.

The institution auditor is a read-only account. The auditor cannot create or update accounts, transfer value, approve requests, synchronize roles, or change configuration.

Before you can complete institution auditor tasks, you must have an active deposit token account. You must be a member of the Institution Auditor application group for the organization.

Typically the institution auditor completes tasks in the following order.

1. Expand the organization's accounts.
2. Select an account.
3. Review or filter the account's transactions.
4. Open transaction details when available.
5. Reconcile pending and terminal states by operation ID.

Audit Policies

Audit policies lets the auditor define review criteria for unusual or important activity. Audit policies do not change ledger data and do not approve or reject transactions. You must enable a policy before you can enter or edit the policy parameters.

Policy	Parameters
Single transfer amount	Choose a comparison such as $\geq$, $\leq$, or $=$, and then enter an amount.
Number of transfers exceeding	Enter a count, and then choose the time window: Day, An Hour, Weekly, or Monthly.
Daily transfer total amount exceeding	Enter the amount threshold.
Any transfers to specific accounts	Select the accounts.
Any transfers from specific accounts	Select the accounts.

Transaction Filters

After you load records from an organization or account history, you can use the following filters to find specific transactions.

Filter	Parameters
Transaction ID	Enter an operation or transaction identifier.
Amount	Select a comparison and enter an amount.
Balance	Select a comparison and enter a balance.
Transaction Type	Select a transaction type.
Initiator	Enter a user or organization value.
Account ID	Enter an account identifier.
Purpose/Memo	Enter the text to search for.
Start Date / End Date	Select the beginning and ending dates.

Selecting a transaction history row opens a read-only Transaction details window, which includes the following fields.

Section	Information Displayed
Summary	Transaction ID, Holding ID, Purpose/Memo, Block Number, Transaction Time, Transaction Type, Amount, and Token ID.

Section	Information Displayed
From	Organization, User ID, and From Account ID. The From party displays - for a creation transaction.
To	Organization, User ID, and To Account ID. The To party displays - for a retirement transaction.

Troubleshooting

If data is inconsistent, refresh the history. If records outside the assigned organization are displayed, report an access control issue. If expected history is not displayed, ask the platform administrator to verify ORDS routes and credentials.

Institution Manager

The institution manager reviews pending participant transfers associated with the organization's manager/escrow account.

The institution manager can approve a direct manager action or complete a transfer after its configured approval sequence is satisfied; it can also reject a pending transfer. Sequential approval-policy steps are handled by the configured bank approver account.

Before you can complete institution manager tasks, you must be a member of the Institution Manager application group for the organization with the required Escrow ledger role.

Typically the institution manager completes tasks in the following order.

1. Select **Pending Transfers**.
2. Review the source, destination, amount, purpose, and operation ID.
3. For an eligible row, select **Approve** or **Complete**, or select **Reject**. Choose multiple eligible requests to approve them in bulk. (You cannot reject multiple requests at a time. Each rejection requires a reason for rejection.)
4. Refresh and confirm the result in the action history.

For a direct manager action, approval completes the pending transfer. For a policy-controlled transfer, the **Complete** command is available only after the configured approval sequence is satisfied. Rejection stops the pending transfer; it does not settle. The request leaves the Pending Transfers window and is displayed in the history with its terminal state.

Troubleshooting

If encounter a problem or error, confirm that the account is active, the request is still pending, and the current manager is authorized for the request. If a row is no longer displayed, check the history before asking the sender to submit the request again.

Institution Officer

The institution officer deposits value to the bank issuer and transfers value to permitted participant officers and users.

The applicable approval policy determines whether the submitted transfer completes immediately or is routed for approval.

Before you can complete institution officer tasks, you must have an active Institution Officer account with an available balance. Recipient and approver accounts must be active.

The following tasks are typical ones that a institution officer completes.

Deposit to the Bank

1. Select **Deposit to Bank Name**, which opens the transfer window for an eligible bank issuer.
2. Select an active bank issuer and then select **Transfer**.
3. Enter a **Quantity** to deposit. Optionally, select **Purpose** and then select one of the configured purposes.
4. Review the **Approver Organisation** and **Approver User ID** context displayed on the form. These fields identify the active escrow context associated with the sending organization; they do not determine the eventual approver for a transfer governed by an approval policy. After the transfer is submitted, the application evaluates the active approval policy. If approval is not required, the transfer can settle accordingly. If approval is required, the transfer remains pending and is routed through the bank approver sequence configured by that policy.
5. Select **Transfer**, and then record the operation ID.

The request is submitted through the approval-aware transfer path. The Besu contract evaluates the active policy after submission: a transfer can settle immediately when no approval is required. If approval is required it remains pending and is routed to the configured bank approver sequence. The bank issuer receives the value only after the transfer settles.

Transfer to a Recipient

1. Select an active officer or user.
2. Enter a **Quantity** within the available balance and limits. Optionally, select **Purpose** and then select one of the configured purposes.
3. Review the **Approver Organisation** and **Approver User ID** context displayed on the form. These identify an active escrow user from the sending organization; select a different user only when the form offers more than one eligible escrow user.
4. Select **Transfer**, and then record the operation ID.

Status, recipient, precision, available balance, daily limits, and approver context are validated. The active approval policy determines whether the transfer completes immediately or remains pending and is routed to the configured approver or approval sequence. When a pending transfer is approved, it appears as completed in history and the sender and recipient balances are updated. A rejection stops the pending transfer; it does not settle.

Troubleshooting

If an account becomes inactive, refresh the recipients. For limit or balance errors, reduce the amount or ask the organization administrator to review the account. Do not resubmit a pending operation.

Institution User

The institution user transfers deposit tokens to permitted participant officers and users and reviews personal transaction history.

Each transfer follows the active amount-based approval policy and either settles immediately or is routed to the configured approval sequence.

The institution user cannot create accounts, change limits or roles, approve transfers, initialize the token, or change setup.

Before you can complete institution user tasks, you must have an active institution User account with an eligible selected wallet and an available balance. There must be at least one active recipient.

The following tasks are typical ones that an institution user completes.

Transfer Tokens

1. Select an active recipient and then select **Transfer**.
2. Enter the **Quantity** to transfer. Optionally, select **Purpose** and then select one of the configured purposes.
3. Review the **Approver Organisation** and **Approver User ID** context displayed on the form. These fields identify an active escrow user from the sending organization; select a different user only when the form offers more than one eligible escrow user.
4. Review the generated operation ID, and then select **Transfer**.
5. Follow the request status. If approval is required, the active policy routes the approval to the configured bank approver sequence.
6. Refresh the page and check the final status in transaction history. Final approval completes the pending transfer and updates the sender and recipient balances after refresh. Rejection stops the pending transfer; it does not settle and the rejected result is recorded in history.

Troubleshooting

For inactive recipient, precision, balance, or limit errors, correct the indicated field or contact the organization administrator. For a pending request, check the operation status and contact the appropriate configured approver or administrator. Do not submit a duplicate request.

Troubleshoot the Deposit Token Application

The following information covers how to troubleshoot problems you might encounter with the application.

Sign-In Returns to an Error Page

For OIDC, verify the issuer, client, exact callback URL, audience, discovery/JWKS reachability, TLS trust, and session configuration. For LDAP, verify the LDAP configuration. If accounts are missing, make sure to preserve the Token Admin and Org Admin role-only administrator sign-in exceptions. Confirm that the user has an eligible active wallet/account for the configured organization and wallet service instance.

No Wallet Is Available

Verify UWaaS reachability, organization mapping, wallet service instance, user identity, and access scope. If multiple wallets are returned, select the wallet for the intended deposit token account.

Account Exists but an Operation Is Unavailable

Check the account status, application group, and ledger role. The bank administrator can use the **Account Actions > Sync Roles** command when authorized. Log out and log in again after a role change, so that the session is rebuilt.

Organization Order Is Incorrect

Reload the live registry and check for founder tag conflicts. Check for missing, duplicate, or non-consecutive sequence values, which the application can reconcile for ordering. Display name does not control order. Multiple founders require manual data repair before the deposit token application can reconcile the registry.

Stale Balance or Pending State

Use the page refresh action and check the operation ID in the history. Confirm Besu and ORDS health before retrying. Do not submit the operation again until the first operation is fails or is in a terminal state.

Approve/Reject Is Unavailable

Confirm that the account is active, the user has the required escrow/manager role, the request is still pending, and the request is assigned to the current approval context.

If a bank approver cannot see or use **Approve** or **Reject**, the bank administrator should verify the following conditions.

- The approver's user account is active.
- The user is configured with the Bank Approver application group/persona.
- The approver has an active/eligible wallet required to participate in the approval flow.
- The organization ID and user ID configured for the active policy's current approval step exactly match the intended bank approver account.
- For a sequential approval policy, the request has reached that approver's configured step; a later approver is not the current approver until the preceding required approval step has been completed.

The availability to approve or reject policy-controlled transfers is determined by the active policy and its current configured bank approver step, not simply by the manager who is associated with the transaction.

Glossary

Index