Deploy a token bridge using the Orbit SDK

TheArbitrum Nitro stack was designed without native support for specific token bridging standards at the protocol level, but Offchain Labs crafted a "canonical bridge" that ensures seamless token transfers between the parent and child chain.

The token bridge architecture includes contracts located on theparent chain as well as a complementary set of contracts on thechild chain . These entities communicate via theRetryable Ticket protocol, ensuring efficient and secure interactions.

UNDER CONSTRUCTION This document is under construction and may change significantly as we incorporatestyle guidance and feedback from readers. Feel free to request specific clarifications by clicking theRequest an update button at the top of this document. info See theERC-20 token bridge overview for more information about the token bridge's design and operational dynamics, and the"create-token-bridge-eth" example for additional guidance.

Token Bridge Deployment Steps

Following thedeployment and initialization of the Orbit chain, the subsequent phase involves deploying contracts on both the parent and child chains.

To establish and configure the token bridge effectively, the process can be broken down into the following steps:

1. Token Approval
2. Token Bridge Contract Deployment
3. Transaction Recipient and Checking for Deployment on Child Chain
4. Deployment Information and Contract Addresses
5. Setting Up the WETH Gateway

info The token bridge deployment process depends on the type of Orbit chain. In the following steps, the main flow is the same for all different kinds of Orbit chains exceptstep 1 , which is only needed forCustom fee token Orbit chains, andstep 5 , just forETH -based Orbit chains.

1. Token Approval (Custom fee token Orbit chains only)

Initiating the deployment of a token bridge forCustom Fee Token on orbit chains begins with ensuring theTokenBridgeCreator contract is granted sufficient approvals of the native token. To facilitate this process, the Orbit SDK provides two essential APIs:

1. createTokenBridgeEnoughCustomFeeTokenAllowance
2. : This API is designed to verify that the deployer's address has enough allowance to pay for the fees associated with the bridge token deployment.
3. createTokenBridgePrepareCustomFeeTokenApprovalTransactionRequest
4. : This API assists in generating the raw transaction required to approve the native token for theTokenBridgeCreator
5. contract.

The following example demonstrates how to leverage these APIs effectively to check for and, if necessary, grant approval to theTokenBridgeCreator contract:

const allowanceParams =

{ nativeToken , owner : rollupOwner . address , publicClient : parentChainPublicClient , } ; if

(! (await

createTokenBridgeEnoughCustomFeeTokenAllowance (allowanceParams)))

{ const approvalTxRequest = await

createTokenBridgePrepareCustomFeeTokenApprovalTransactionRequest (allowanceParams) ; } In this scenario,allowanceParams includes the native token details, the roll-up owner's address, and the public client for the parent chain. Initially, thecreateTokenBridgeEnoughCustomFeeTokenAllowance API checks if the deployer's allowance is sufficient. If the allowance is inadequate, thecreateTokenBridgePrepareCustomFeeTokenApprovalTransactionRequest API is called to create the necessary approval transaction.

It is important to note that after generating the raw transaction, the deployer must sign and broadcast it to the network to finalize the approval process.

2. Token Bridge Contract Deployment

The deployment of token bridge contracts constitutes the foundational step in establishing a bridge between the parent and the Orbit chain. This process mirrors the deployment methodology used for orbit chain contracts, where a primary contract, namedRollupCreator , facilitates the deployment of core contracts. In the context of token bridge contracts, theTokenBridgeCreator contract assumes a similar pivotal role by orchestrating the deployment across both the parent and the Orbit chain. Feel free to take a look at theTokenBridgeCreator Solidity code .

TokenBridgeCreator can deploy the token bridge contracts on both the parent and child chains in a single transaction. A common query is how it manages to directly deploy contracts on the child chain from the parent chain. The solution lies in using the Retryable Tickets protocol, which facilitates the transmission of the deployment message between the two chains. This message, once transmitted, triggers the contract deployment by the Retryable Tickets mechanism. For an in-depth understanding, please refer to thisexplanation of the Retryable Ticket system .

To streamline the deployment process, an API has been integrated into our Orbit SDK, designed to automate the deployment by interacting with theTokenBridgeCreator contract. The API iscreateTokenBridgePrepareTransactionRequest , which processes the necessary inputs and generates a transaction request tailored for token bridge deployment. Below is an illustrative example of how to use this API:

const txRequest =

await

createTokenBridgePrepareTransactionRequest ({ params :

{ rollup : rollupContractAddress , rollupOwner : rollupOwnerAddress , } , parentChainPublicClient , orbitChainPublicClient , account : rollupOwnerAddress , }) ; In the above example,rollupContractAddress refers to the Orbit chain's rollup contract address, androllupOwnerAddress denotes the rollup owner's address. Additionally,parentChainPublicClient andorbitChainPublicClient represent the public clients for the parent and Orbit chains, respectively, as defined by Viem. For detailed insights into their configuration and usage, consider exploring thistoken bridge deployment example .

Following the creation of the raw transaction, the next steps involve signing it and broadcasting it to the relevant blockchain network to complete the deployment process.

3. Transaction Recipient and Checking for Deployment on Child Chain

Following the dispatch of the deployment transaction, retrieving the transaction receipt and verifying the successful deployment of the contracts on both the parent and child chains is crucial. Our Orbit SDK includes a dedicated API for this purpose, namedcreateTokenBridgePrepareTransactionReceipt , which simplifies the process of obtaining the deployment transaction's recipient. An illustrative use of this API is as follows:

const txReceipt =

createTokenBridgePrepareTransactionReceipt (await parentChainPublicClient . waitForTransactionReceipt ({

hash : txHash }) ,) ; In this scenario,txHash represents the hash of the deployment transaction initiated in the previous step. ThewaitForTransactionReceipt API from Viem captures the transaction's recipient on the parent chain. ThecreateTokenBridgePrepareTransactionReceipt API enhances the basic functionality provided by Viem'swaitForTransactionReceipt , introducing a specialized method namedwaitForRetryables to handle the outcome (in this case,txReceipt).

By employing thewaitForRetryables method, one can ascertain the success of Retryable Tickets on the parent chain. Here is how to use this API effectively:

const orbitChainRetryableReceipts =

await txReceipt . waitForRetryables ({ orbitPublicClient : orbitChainPublicClient , }) ;

if

(orbitChainRetryableReceipts [0] . status

!==

'success')

{ throw

new

Error (Retryable status is not success: { orbitChainRetryableReceipts [0] . status } . Aborting... ,) ; }

console . log (Retryable executed successfully) ; In this example, thewaitForRetryables method is invoked on thetxReceipt to monitor the execution of Retryable Tickets and verify their status. Asuccess status indicates that the Retryable Tickets have been executed successfully, ensuring the contracts' deployment. It's important to note that this process involves two Retryable Tickets. You can check out amore comprehensive walkthrough of the example. This enhanced approach not only simplifies the retrieval of transaction receipts but also provides a reliable method for verifying contract deployment across chains.

4. Deployment Information and Contract Addresses

Once we are done with the deployment and are assured that the Retryable Tickets are successful, it's time to get the deployment information and all token bridge contract addresses. To get this information, we have an API on Orbit SDK namedgetTokenBridgeContracts . You can use this method to retrieve the data for the token bridge recipient.

Here's an example of how to get the contract addresses from thetxReceipt generated in the previous steps:

const tokenBridgeContracts =

await txReceipt . getTokenBridgeContracts ({ parentChainPublicClient , }) ;

5. Setting Up the WETH Gateway (ETH-based Orbit chains only)

The last step in spinning up the token bridge for an ETH-based Orbit chain consists of setting up theWETH Gateway .

note That step only applies to ETH-based Orbit chains, not Custom fee token orbit chains. Our canonical bridge design has a separate custom gateway for WETH to bridge it in and out of the Orbit chain.

You can find more info about WETH gateways in our"other gateways flavors" documentation . So, after the token bridge has been deployed and you have secured a successful deployment on both parent and child chains, it's time to set theWETH Gateway on both parent and child chains. To handle that, we have two APIs on our Orbit SDK:

1.createTokenBridgePrepareSetWethGatewayTransactionRequest

:

This API helps you create the raw transaction, which handles the WETH gateway on both parent and child chains.

Here's an example of how to use this API:

const setWethGatewayTxRequest =

await

createTokenBridgePrepareSetWethGatewayTransactionRequest ({ rollup : rollupContractAddress , parentChainPublicClient , orbitChainPublicClient , account : rollupOwnerAddress , retryableGasOverrides :

{ gasLimit :

{ percentIncrease :

200n , } , } , }) ; In this examplerollupContractAddress is the address of Orbit chain's rollup contract,rollupOwnerAddress is the address of rollup owner,parentChainPublicClient andorbitChainPublicClient are the parent and orbit chain public clients. Also this API has optional fields to override the Retryable ticket setups. In this examplepercentIncrease is the buffer to increase the gas limit for the retryable ticket to be sure about the success of the ticket. After creating the raw transaction you need to use Viem to sign and broadcast the transaction to the network.

2.createTokenBridgePrepareSetWethGatewayTransactionReceipt

After sending the transaction, you need get the recept of the transaction to be able to check about the success of the Retryable Tickets created on step 1, which is going to set WETH gateway on the Orbit chain. To do that we are usingcreateTokenBridgePrepareSetWethGatewayTransactionReceipt API and alsowaitForRetryables method of it to check for the retryable ticket status. For the example in this doc we can use this API as follow:

const setWethGatewayTxReceipt =

createTokenBridgePrepareSetWethGatewayTransactionReceipt (await parentChainPublicClient . waitForTransactionReceipt ({

hash : setWethGatewayTxHash }) ,) ; const orbitChainSetWethGatewayRetryableReceipt =

await setWethGatewayTxReceipt . waitForRetryables ({ orbitPublicClient : orbitChainPublicClient , }) ; if

(orbitChainSetWethGatewayRetryableReceipt [0] . status

!==

'success')

{ throw

new

Error (Retryable status is not success: { orbitChainSetWethGatewayRetryableReceipt [0] . status } . Aborting... ,) ; console . log (Retryables executed successfully) ; In this examplesetWethGatewayTxHash is the hash of the transaction you sent to set the WETH gateway. Edit this page Last updatedonApr 2, 2024 Previous Configure your chain's node Next Configure your chain