**DSCTransferHub API Documentation (avalanche\_dsc\_transfer\_hub.py)**

**Overview**

The avalanche\_dsc\_transfer\_hub.py script is a FastAPI application that provides a RESTful API for interacting with the DSCTransferHub smart contract on the Avalanche Fuji testnet (chain ID 43113). The contract enables cross-chain token transfers (e.g., DSC) to Ethereum Sepolia (chain selector 16015286601757825753) using Chainlink CCIP. It supports depositing tokens and AVAX, transferring tokens cross-chain, and withdrawing token deposits. The API is designed for secure blockchain interactions and uses environment variables for configuration.

This document is intended for:

* **Frontend Engineers**: To integrate the API into user interfaces, understand endpoint inputs, outputs, and error handling.
* **AI Engineers**: To understand contract interactions, extend cross-chain functionality, and debug issues like transaction reverts or library errors.

**Setup**

**Prerequisites**

* **Python**: Version 3.8 or higher.
* **Dependencies**: Requires fastapi, uvicorn, web3, python-dotenv, and pydantic. Ensure web3.py is version 6.0.0 or higher to avoid errors like missing attributes.
* **Avalanche Fuji Node**: Access to a Fuji RPC endpoint (e.g., https://api.avax-test.network/ext/bc/C/rpc).
* **Environment Variables**: Create a .env file in the project root with:
  + AVALANCHE\_RPC\_URL: Fuji node endpoint (e.g., https://api.avax-test.network/ext/bc/C/rpc).
  + PRIVATE\_KEY: Avalanche private key (with 0x prefix) for signing transactions.
  + AVALANCHE\_DSC\_TRANSFER\_HUB\_ADDRESS: DSCTransferHub contract address on Fuji.

**Running the API**

1. Save the script as avalanche\_dsc\_transfer\_hub.py in your project directory (e.g., /mnt/d/chainlink-CrossChain-Stablecoin/Base-Hackathon-Stable-Token/dsc-api).
2. Start the FastAPI server using the command: uvicorn avalanche\_dsc\_transfer\_hub:app --host 0.0.0.0 --port 8000 --reload.
3. Access the API at http://localhost:8000. Use /docs for the interactive Swagger UI to test endpoints.

**Avoiding Common Errors**

* **Missing Attributes**: Ensure no local file named web3.py or Web3.py exists in the project directory, as it can conflict with web3.py. Verify web3.py version is 6.0.0 or higher.
* **Contract Not Deployed**: Confirm the DSCTransferHub contract is deployed at the specified address using a blockchain explorer like Snowtrace.
* **Nonce Issues**: The API retries transactions up to three times if a "nonce too low" error occurs, common in Fuji due to network congestion.
* **Network Connection**: Ensure the Fuji RPC endpoint is accessible and the chain ID is 43113.

**Endpoints**

**1. Approve Tokens (/approve-tokens)**

* **Method**: POST
* **Description**: Approves the DSCTransferHub contract to spend tokens (e.g., DSC) on behalf of the user’s wallet. This is a prerequisite for depositing tokens for cross-chain transfers.
* **Request Body**:
  + token\_address: The address of the token to approve (e.g., DSC contract address on Fuji).
  + amount: The amount of tokens to approve (in token units, e.g., 1.0 for 1 DSC).
  + Example:
  + {
  + "token\_address": "0x...DSC\_ADDRESS...",
  + "amount": 1.0
  + }
* **Response**:
  + Success (approval needed): Returns the transaction hash of the approval.
  + {"approval\_tx\_hash": "0x..."}
  + Success (no approval needed): Indicates sufficient allowance exists.
  + {"message": "No approval needed; sufficient allowance already set"}
  + Error: Details the issue (e.g., invalid address or unsupported token).
  + {"detail": "Token not supported by DSCTransferHub"}
* **Status Codes**:
  + 200: Approval succeeded or was not needed.
  + 400: Invalid token address, non-positive amount, or token not supported by DSCTransferHub.
  + 500: Transaction failure or server error (e.g., network issue).
* **Notes**:
  + Checks if the token is supported by DSCTransferHub before approving.
  + Verifies existing allowance to avoid redundant transactions.
  + Must be called before /deposit-tokens or /transfer-tokens.

**2. Deposit Tokens (/deposit-tokens)**

* **Method**: POST
* **Description**: Deposits tokens (e.g., DSC) into the DSCTransferHub contract for cross-chain transfer preparation to Sepolia. Tokens must be approved first.
* **Request Body**:
  + token\_address: The address of the token to deposit (e.g., DSC contract address on Fuji).
  + amount: The amount of tokens to deposit (in token units, e.g., 0.5 for 0.5 DSC).
  + Example:
  + {
  + "token\_address": "0x...DSC\_ADDRESS...",
  + "amount": 0.5
  + }
* **Response**:
  + Success: Returns the transaction hash of the deposit.
  + {"tx\_hash": "0x..."}
  + Error: Details the issue (e.g., insufficient allowance).
  + {"detail": "Insufficient allowance. Approve tokens first"}
* **Status Codes**:
  + 200: Deposit succeeded.
  + 400: Invalid token address, non-positive amount, insufficient allowance, or unsupported token.
  + 500: Transaction revert (e.g., contract logic error) or server error.
* **Notes**:
  + Requires prior approval via /approve-tokens.
  + Verifies token support using DSCTransferHub’s isSupportedToken function.
  + Deposits are tracked in userTokenDeposits for the user and token.

**3. Send AVAX (/send-avax)**

* **Method**: POST
* **Description**: Sends AVAX to the DSCTransferHub contract to deposit funds for cross-chain transfer fees, triggering the contract’s receive() function.
* **Request Body**:
  + amount: The amount of AVAX to send (in AVAX units, e.g., 0.1 for 0.1 AVAX).
  + Example:
  + {
  + "amount": 0.1
  + }
* **Response**:
  + Success: Returns transaction details, including balances and events.
  + {
  + "tx\_hash": "0x...",
  + "avax\_balance\_before": 1.5,
  + "avax\_balance\_after": 1.39,
  + "avax\_deposits": 0.1,
  + "events": {
  + "deposit\_events": [{"from": "0x...", "token": "0x...", "amount": 0.1}]
  + },
  + "tx\_data": "0x"
  + }
  + Error: Details the issue (e.g., insufficient AVAX).
  + {"detail": "Insufficient AVAX balance: 1.0 AVAX available, 1.1 AVAX required"}
* **Status Codes**:
  + 200: AVAX deposit succeeded.
  + 400: Non-positive amount or insufficient AVAX balance (including gas).
  + 500: Transaction revert or server error.
* **Notes**:
  + Checks AVAX balance before sending, accounting for gas costs.
  + Parses TokensDeposited events to confirm deposit.
  + Uses a gas limit of 100,000 and doubled gas price for reliability.
  + AVAX deposits are tracked in userEthDeposits.

**4. Transfer Tokens (/transfer-tokens)**

* **Method**: POST
* **Description**: Transfers tokens cross-chain to a receiver on Ethereum Sepolia (chain selector 16015286601757825753) using Chainlink CCIP. Requires sufficient token and AVAX deposits.
* **Request Body**:
  + receiver: The recipient’s address on Sepolia.
  + token\_address: The address of the token to transfer (e.g., DSC on Fuji).
  + amount: The amount of tokens to transfer (in token units, e.g., 0.5).
  + Example:
  + {
  + "receiver": "0x...SEPOLIA\_ADDRESS...",
  + "token\_address": "0x...DSC\_ADDRESS...",
  + "amount": 0.5
  + }
* **Response**:
  + Success: Returns the transaction hash of the transfer.
  + {"tx\_hash": "0x..."}
  + Error: Details the issue (e.g., insufficient deposits).
  + {"detail": "Insufficient token deposit"}
* **Status Codes**:
  + 200: Transfer succeeded.
  + 400: Invalid receiver or token address, non-positive amount, unsupported token, unallowlisted chain, or insufficient token/AVAX deposits.
  + 500: Transaction revert or server error.
* **Notes**:
  + Validates token support and whether Sepolia is allowlisted.
  + Checks userTokenDeposits and userEthDeposits for sufficient funds.
  + Triggers TokensTransferred event, which includes the CCIP message ID.

**5. Withdraw Deposits (/withdraw-deposits)**

* **Method**: POST
* **Description**: Withdraws token deposits from DSCTransferHub to the user’s wallet. AVAX withdrawals are not supported by this endpoint.
* **Request Body**:
  + token\_address: The address of the token to withdraw (e.g., DSC on Fuji). Optional, but currently required for token withdrawals.
  + Example:
  + {
  + "token\_address": "0x...DSC\_ADDRESS..."
  + }
* **Response**:
  + Success: Returns the transaction hash of the withdrawal.
  + {"tx\_hash": "0x..."}
  + Error: Details the issue (e.g., no deposits).
  + {"detail": "Nothing to withdraw"}
* **Status Codes**:
  + 200: Withdrawal succeeded.
  + 400: Invalid token address, unsupported token, or no deposits available.
  + 500: Transaction revert or server error.
* **Notes**:
  + Checks userTokenDeposits to confirm available balance.
  + Does not support AVAX withdrawals in this implementation.

**Frontend Integration**

**Example Request Format**

To deposit tokens for cross-chain transfer:

* **Endpoint**: /deposit-tokens
* **Request**:
* {
* "token\_address": "0x...DSC\_ADDRESS...",
* "amount": 0.5
* }
* **Integration**:
  + Use a library like axios or fetch to send POST requests.
  + Validate addresses client-side (e.g., using ethers.js to check Avalanche address format).
  + Display the transaction hash and link to Snowtrace (https://testnet.snowtrace.io/tx/<tx\_hash>).

**Error Handling**

* **400 Errors**: Validate inputs (addresses, amounts) before sending. Display messages like “Invalid address” or “Insufficient deposits” to users.
* **500 Errors**: Show transaction revert reasons (e.g., “Contract execution reverted: Token not supported”). Log details for debugging.
* **Timeouts**: Account for transaction times up to 300 seconds. Show loading indicators during processing.

**UI Considerations**

* **Form Validation**: Ensure addresses are valid Avalanche addresses and amounts are positive. Use libraries like ethers.js for validation.
* **Loading States**: Display spinners for transaction endpoints (/deposit-tokens, /send-avax, /transfer-tokens, /withdraw-deposits).
* **Balance Display**: Query userTokenDeposits and userEthDeposits via contract calls to show available deposits.
* **Cross-Chain Feedback**: For /transfer-tokens, inform users that tokens are sent to Sepolia and may require action on Sepolia (e.g., via evm\_dsc\_transfer\_hub.py’s /receive-dsc if implemented).
* **Transaction Links**: Provide Snowtrace links for transaction hashes to track status.

**AI Engineer Notes**

**Contract Interactions**

* **Purpose**: The DSCTransferHub contract manages token and AVAX deposits, cross-chain transfers to Sepolia via Chainlink CCIP, and token withdrawals.
* **Functions Used**:
  + depositTokens: Deposits tokens for cross-chain transfer.
  + transferTokens: Sends tokens to Sepolia with a specified receiver.
  + withdrawDeposits: Withdraws token deposits.
  + userTokenDeposits and userEthDeposits: Query deposit balances.
  + isSupportedToken: Checks if a token is supported.
  + allowlistedChains: Verifies if Sepolia is allowlisted.
* **Events**:
  + TokensDeposited: Emitted on token or AVAX deposits.
  + TokensTransferred: Emitted on cross-chain transfers, including CCIP message ID.

**Cross-Chain Mechanics**

* **Workflow**:
  1. Approve tokens for DSCTransferHub (/approve-tokens).
  2. Deposit tokens (/deposit-tokens) and AVAX for fees (/send-avax).
  3. Transfer tokens to Sepolia (/transfer-tokens).
  4. On Sepolia, use a corresponding endpoint (e.g., /receive-dsc in evm\_dsc\_transfer\_hub.py, if implemented) to receive tokens.
* **Chainlink CCIP**: The contract uses CCIP with a router and LINK token address set in the constructor. The Sepolia chain selector (16015286601757825753) is hardcoded.
* **Coordination**: Ensure the Sepolia receiver has access to a receiving endpoint (e.g., /receive-dsc) with sufficient DSC allowance.

**Debugging Tips**

* **Transaction Reverts**: Use Snowtrace to check revert reasons or simulate calls to decode errors (e.g., “Token not supported” or “Destination chain not allowlisted”).
* **Contract Deployment**: Verify DSCTransferHub is deployed at the specified address using Snowtrace.
* **Gas Estimation**: Fuji may require high gas prices (e.g., ≥25 nAVAX). The API doubles the gas price for reliability.
* **Cross-Chain Issues**: Confirm Sepolia is allowlisted in DSCTransferHub. Check CCIP configuration (router and LINK token addresses).
* **Attribute Errors**: Ensure no conflicting web3.py file exists and web3.py is version 6.0.0 or higher.

**Testing**

**Setup**

1. Fund the account with Fuji AVAX using a faucet (e.g., https://faucet.avax.network/).
2. Backup the script before testing to avoid accidental overwrites.
3. Ensure the DSCTransferHub contract address is valid and deployed.

**Test Endpoint**

* **Command**: Test the /send-avax endpoint to deposit AVAX.
* curl -X POST http://localhost:8000/send-avax -H 'Content-Type: application/json' -d '{"amount": 0.01}'
* **Expected Response**:
* {
* "tx\_hash": "0x...",
* "avax\_balance\_before": 1.0,
* "avax\_balance\_after": 0.989,
* "avax\_deposits": 0.01,
* "events": {...},
* "tx\_data": "0x"
* }

**Verify on Snowtrace**

* Check the transaction status on https://testnet.snowtrace.io/tx/<tx\_hash>.

**Check Logs**

* Monitor server logs for errors or transaction details using a command like tail -f uvicorn.log.

**Security Considerations**

* **Private Key**: Store the private key securely in the .env file, never in source code.
* **Gas Limits**: The API uses 100,000 gas for transfers and 3,000,000 for contract calls. Adjust if transactions fail due to gas shortages.
* **Input Validation**: All endpoints validate addresses, amounts, and token support to prevent invalid transactions.
* **Cross-Chain Security**: Ensure DSCTransferHub is audited for CCIP integration. Verify allowlisted chains and token support.
* **Rate Limiting**: Consider adding middleware to prevent API abuse, especially for transaction-heavy endpoints.

**Cross-Chain Notes**

* The /transfer-tokens endpoint sends tokens to Sepolia, where they can be received via a corresponding endpoint (e.g., /receive-dsc in evm\_dsc\_transfer\_hub.py, if implemented).
* The process requires:
  + Sufficient token deposits (/deposit-tokens).
  + AVAX deposits for CCIP fees (/send-avax).
  + Coordination with the Sepolia receiver to call a receiving endpoint with the correct Fuji chain ID (43113).
* Verify the CCIP chain selector for Sepolia (16015286601757825753) in the contract configuration.

**Verify Transaction at Chainlink Explorer**

https://ccip.chain.link/