Token

→ Anyone can issue various types of tokens in the XRP Ledger, ranging from informal "IOUs" to fiat-backed stablecoins, purely digital fungible and semi-fungible tokens, and more. This tutorial shows the technical steps of creating a token in the ledger. For more information on how XRP Ledger tokens work, see Issued Currencies; for more on the business decisions involved in issuing a stablecoin, see Become an XRP Ledger Gateway.

Steps

1. Get Credentials
   → To transact on the XRP Ledger, you need an address and secret key, and some XRP. You also need one or more recipients who are willing to hold the tokens you issue: unlike in some other blockchains, in the XRP Ledger you cannot force someone to hold a token they do not want.

   2. Connect to the Network

→ You must be connected to the network to submit transactions to it. The following code shows how to connect to a public XRP Ledger Testnet server with a supported client library.

```
// In browsers, use a <script> tag. In Node.js, uncomment the following line:

// const xrpl = require('xrpl')

// Wrap code in an async function so we can use await

async function main() {

// Define the network client

const client = new xrpl.Client("wss://s.altnet.rippletest.net:51233")

await client.connect()

// ... custom code goes here

// Disconnect when done (If you omit this, Node.js won't end the process)

client.disconnect()

}

main()
```

3. Configure issuer settings

→ First, configure the settings for your cold address (which will become the issuer of your token). Most settings can be reconfigured later, with the following exceptions:

Default Ripple: This setting is required so that users can send your token to each other. It's best to enable it before setting up any trust lines or issuing any tokens.

Authorized Trust Lines: (Optional) This setting (also called "Require Auth") limits your tokens to being held only by accounts you've explicitly approved. You cannot enable this setting if you already have any trust lines or offers for any token. Note: To use authorized trust lines, you must perform additional steps that are not shown in this tutorial.

```
// Configure issuer (cold address) settings ----------------------------------

const cold_settings_tx = {

"TransactionType": "AccountSet",

"Account": cold_wallet.address,

"TransferRate": 0,

"TickSize": 5,

"Domain": "6578616D706C652E636F6D", // "example.com"

"SetFlag": xrpl.AccountSetAsfFlags.asfDefaultRipple,

// Using tf flags, we can enable more flags in one transaction

"Flags": (xrpl.AccountSetTfFlags.tfDisallowXRP |

xrpl.AccountSetTfFlags.tfRequireDestTag)

}

const cst_prepared = await client.autofill(cold_settings_tx)

const cst_signed = cold_wallet.sign(cst_prepared)

console.log("Sending cold address AccountSet transaction...")

const cst_result = await client.submitAndWait(cst_signed.tx_blob)

if (cst_result.result.meta.TransactionResult == "tesSUCCESS") {

console.log(`Transaction succeeded: https://testnet.xrpl.org/transactions/${cst_signed.hash}`)

} else {

throw `Error sending transaction: ${cst_result}`

}
```

4. Wait for validation

→ Most transactions are accepted into the next ledger version after they're submitted, which means it may take 4-7 seconds for a transaction's outcome to be final. You should wait for your earlier transactions to be fully validated before proceeding to the later steps, to avoid unexpected failures from things executing out of order. For more information, see Reliable Transaction Submission.

5. Configure hot address settings

→ The following code sample shows how to send an AccountSet transaction to enable the recommended hot address settings.

```
// Configure hot address settings ------------------------------------------

const hot_settings_tx = {

"TransactionType": "AccountSet",

"Account": hot_wallet.address,

"Domain": "6578616D706C652E636F6D", // "example.com"

// enable Require Auth so we can't use trust lines that users

// make to the hot address, even by accident:

"SetFlag": xrpl.AccountSetAsfFlags.asfRequireAuth,

"Flags": (xrpl.AccountSetTfFlags.tfDisallowXRP |

xrpl.AccountSetTfFlags.tfRequireDestTag)

}

const hst_prepared = await client.autofill(hot_settings_tx)

const hst_signed = hot_wallet.sign(hst_prepared)

console.log("Sending hot address AccountSet transaction...")

const hst_result = await client.submitAndWait(hst_signed.tx_blob)

if (hst_result.result.meta.TransactionResult == "tesSUCCESS") {

console.log(`Transaction succeeded: https://testnet.xrpl.org/transactions/${hst_signed.hash}`)

} else {

throw `Error sending transaction: ${hst_result.result.meta.TransactionResult}`
}
```

6. Wait for validation

→ As before, wait for the previous transaction to be validated by consensus before continuing.

7. Create trust line from hot to cold address

→ Before you can receive tokens, you need to create a trust line to the token issuer. This trust line is specific to the currency code of the token you want to issue, such as USD or FOO. You can choose any currency code you want; each issuer's tokens are treated as separate in the XRP Ledger protocol. However, users' balances of tokens with the same currency code can ripple between different issuers if the users enable rippling settings.

```
// Create trust line from hot to cold address --------------------------------

const currency_code = "FOO"

const trust_set_tx = {

"TransactionType": "TrustSet",

"Account": hot_wallet.address,

"LimitAmount": {

"currency": currency_code,

"issuer": cold_wallet.address,

"value": "10000000000" // Large limit, arbitrarily chosen

}

}

const ts_prepared = await client.autofill(trust_set_tx)

const ts_signed = hot_wallet.sign(ts_prepared)

console.log("Creating trust line from hot address to issuer...")

const ts_result = await client.submitAndWait(ts_signed.tx_blob)

if (ts_result.result.meta.TransactionResult == "tesSUCCESS") {

console.log(`Transaction succeeded: https://testnet.xrpl.org/transactions/${ts_signed.hash}`)
} else {

throw `Error sending transaction: ${ts_result.result.meta.TransactionResult}`

}
```

8. Wait for validation

→ As before, wait for the previous transaction to be validated by consensus before continuing.

9. Send token

→ Now you can create tokens by sending a Payment transaction from the cold address to the hot address. This transaction should have the following attributes (dot notation indicates nested fields):

Field	Value
TransactionType	"Payment"
Account	The cold address issuing the token.
Amount	An token amount specifying how much of which token to create.
Amount.currency	The currency code of the token.
Amount.value	Decimal amount of the token to issue, as a string.
Amount.issuer	The cold address issuing the token.
Destination	The hot address (or other account receiving the token)
Paths	Omit this field when issuing tokens.
SendMax	Omit this field when issuing tokens.
DestinationTag	Any whole number from 0 to 232-1. You must specify something here if you enabled Require Destination Tags on the hot address.

```
// Send token -------------------------------------------------------------

const issue_quantity = "3840"

const send_token_tx = {

"TransactionType": "Payment",

"Account": cold_wallet.address,
"Amount": {

"currency": currency_code,

"value": issue_quantity,

"issuer": cold_wallet.address

},

"Destination": hot_wallet.address,

"DestinationTag": 1 // Needed since we enabled Require Destination Tags

// on the hot account earlier.

}

const pay_prepared = await client.autofill(send_token_tx)

const pay_signed = cold_wallet.sign(pay_prepared)

console.log(`Sending ${issue_quantity} ${currency_code} to ${hot_wallet.address}...`)

const pay_result = await client.submitAndWait(pay_signed.tx_blob)

if (pay_result.result.meta.TransactionResult == "tesSUCCESS") {

console.log(`Transaction succeeded:
https://testnet.xrpl.org/transactions/${pay_signed.hash}`)

} else {

throw `Error sending transaction: ${pay_result.result.meta.TransactionResult}`

}
```

10. Wait for validation
⟶ As before, wait for the previous transaction to be validated by consensus before continuing.

11. Confirm token balances
→ You can check the balances of your token from the perspective of either the token issuer or the hot address. Tokens issued in the XRP Ledger always have balances that sum to 0: negative from the perspective of the issuer and positive from the perspective of the holder.

```
// Check balances ------------------------------------------------------------

console.log("Getting hot address balances...")
```

```
const hot_balances = await client.request({

command: "account_lines",

account: hot_wallet.address,

ledger_index: "validated"

})

console.log(hot_balances.result)

console.log("Getting cold address balances...")

const cold_balances = await client.request({

command: "gateway_balances",

account: cold_wallet.address,

ledger_index: "validated",

hotwallet: [hot_wallet.address]

})

console.log(JSON.stringify(cold_balances.result, null, 2))

client.disconnect()

}
```

→ Now that you've created the token, you can explore how it fits into features of the XRP Ledger
Send tokens from the hot address to other users.

Trade it in the decentralized exchange.

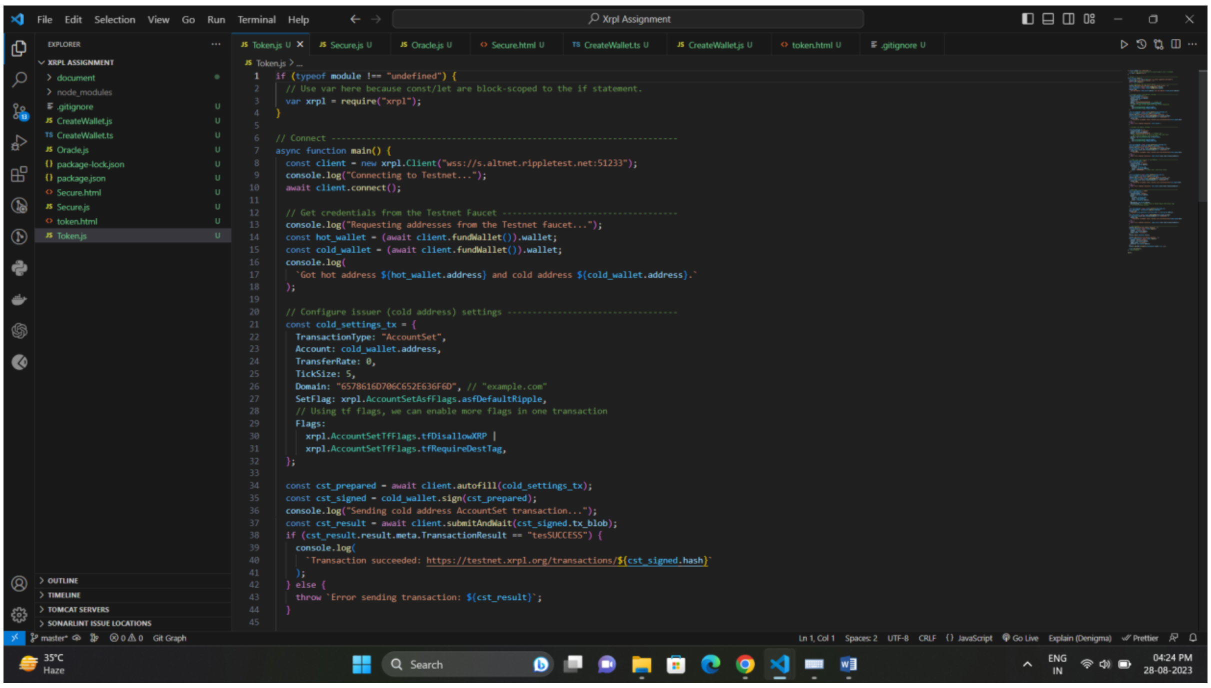
Monitor for incoming payments of your token.

*** <u>Steps for create a XRPL Token</u>**

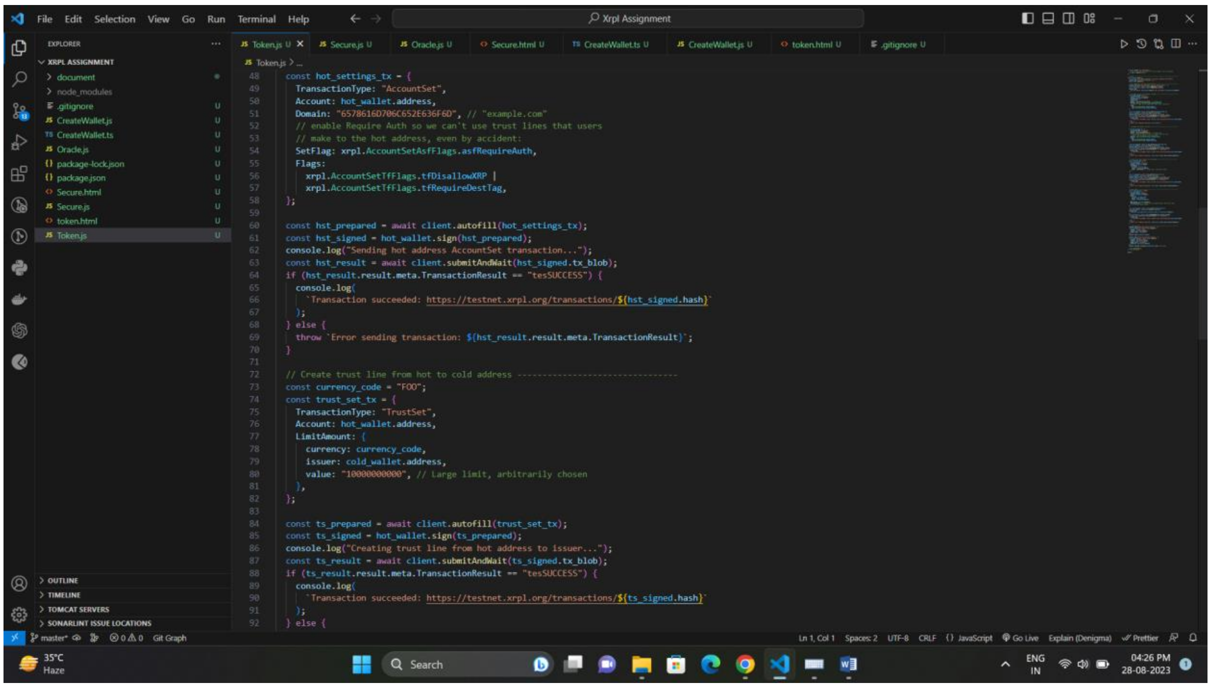
1. Get credentials to transact on the XRP Ledger, you need an address and secret key and some XRP.

2. Connect to the network.
3. Configure issuer settings.
4. Wait for validation.
5. Configure hot address settings.
6. Wait for validation.
7. Create trust line from hot to cold address.

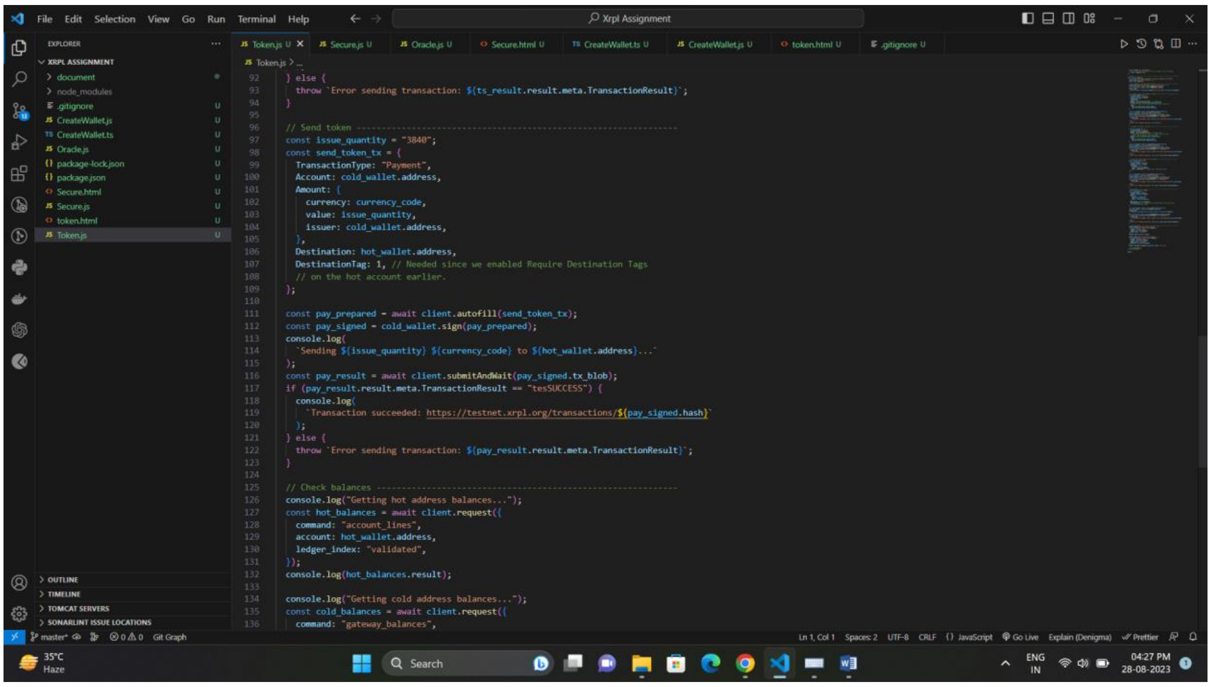
Code snippets



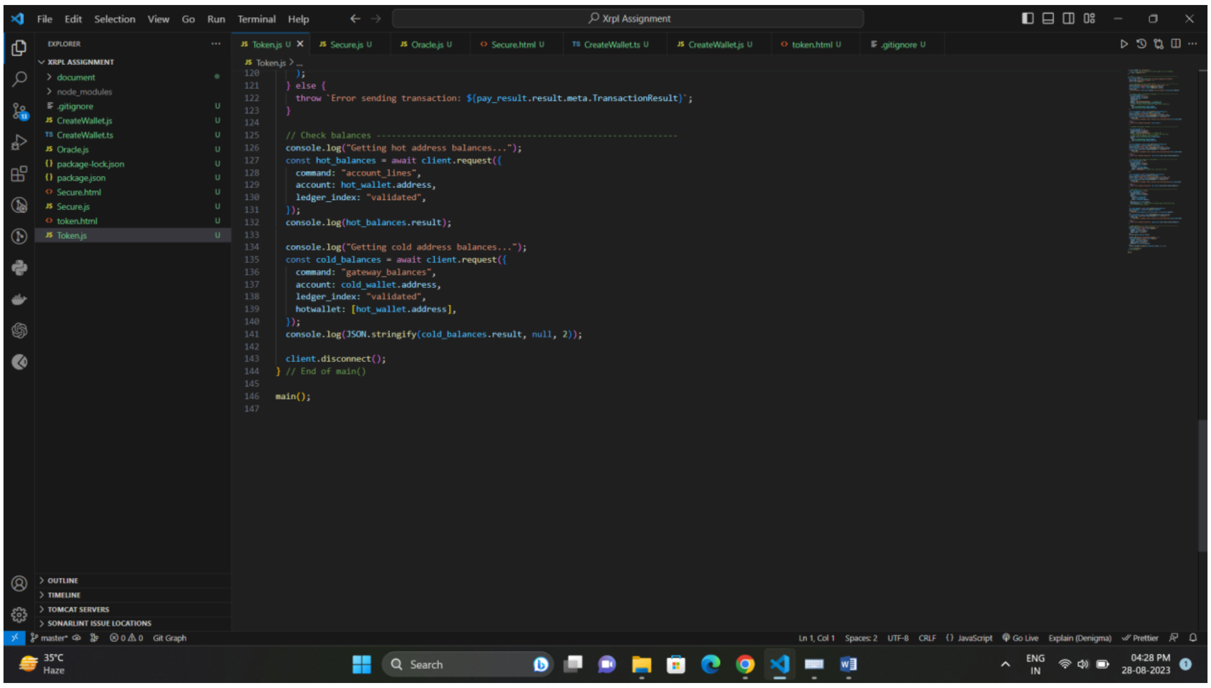
```javascript
if (typeof module !== "undefined") {
  // Use var here because const/let are block-scoped to the if statement.
  var xrpl = require("xrpl");
}

// Connect ---------------------------------------------------------------
async function main() {
  const client = new xrpl.Client("wss://s.altnet.rippletest.net:51233");
  console.log("Connecting to Testnet...");
  await client.connect();

  // Get credentials from the Testnet Faucet ------------------------------
  console.log("Requesting addresses from the Testnet faucet...");
  const hot_wallet = (await client.fundWallet()).wallet;
  const cold_wallet = (await client.fundWallet()).wallet;
  console.log(
    `Got hot address ${hot_wallet.address} and cold address ${cold_wallet.address}.`
  );

  // Configure issuer (cold address) settings -----------------------------
  const cold_settings_tx = {
    TransactionType: "AccountSet",
    Account: cold_wallet.address,
    TransferRate: 0,
    TickSize: 5,
    Domain: "6578616D706C652E636F6D", // "example.com"
    SetFlag: xrpl.AccountSetAsfFlags.asfDefaultRipple,
    // Using tf flags, we can enable more flags in one transaction
    Flags:
      xrpl.AccountSetTfFlags.tfDisallowXRP |
      xrpl.AccountSetTfFlags.tfRequireDestTag,
  };

  const cst_prepared = await client.autofill(cold_settings_tx);
  const cst_signed = cold_wallet.sign(cst_prepared);
  console.log("Sending cold address AccountSet transaction...");
  const cst_result = await client.submitAndWait(cst_signed.tx_blob);
  if (cst_result.result.meta.TransactionResult == "tesSUCCESS") {
    console.log(
      `Transaction succeeded: https://testnet.xrpl.org/transactions/${cst_signed.hash}`
    );
  } else {
    throw `Error sending transaction: ${cst_result}`;
  }

```



```javascript
  const hot_settings_tx = {
    TransactionType: "AccountSet",
    Account: hot_wallet.address,
    Domain: "6578616D706C652E636F6D", // "example.com"
    // enable Require Auth so we can't use trust lines that users
    // make to the hot address, even by accident:
    SetFlag: xrpl.AccountSetAsfFlags.asfRequireAuth,
    Flags:
      xrpl.AccountSetTfFlags.tfDisallowXRP |
      xrpl.AccountSetTfFlags.tfRequireDestTag,
  };

  const hst_prepared = await client.autofill(hot_settings_tx);
  const hst_signed = hot_wallet.sign(hst_prepared);
  console.log("Sending hot address AccountSet transaction...");
  const hst_result = await client.submitAndWait(hst_signed.tx_blob);
  if (hst_result.result.meta.TransactionResult == "tesSUCCESS") {
    console.log(
      `Transaction succeeded: https://testnet.xrpl.org/transactions/${hst_signed.hash}`
    );
  } else {
    throw `Error sending transaction: ${hst_result.result.meta.TransactionResult}`;
  }

  // Create trust line from hot to cold address ---------------------------
  const currency_code = "FOO";
  const trust_set_tx = {
    TransactionType: "TrustSet",
    Account: hot_wallet.address,
    LimitAmount: {
      currency: currency_code,
      issuer: cold_wallet.address,
      value: "10000000000", // Large limit, arbitrarily chosen
    },
  };

  const ts_prepared = await client.autofill(trust_set_tx);
  const ts_signed = hot_wallet.sign(ts_prepared);
  console.log("Creating trust line from hot address to issuer...");
  const ts_result = await client.submitAndWait(ts_signed.tx_blob);
  if (ts_result.result.meta.TransactionResult == "tesSUCCESS") {
    console.log(
      `Transaction succeeded: https://testnet.xrpl.org/transactions/${ts_signed.hash}`
    );
  } else {
```

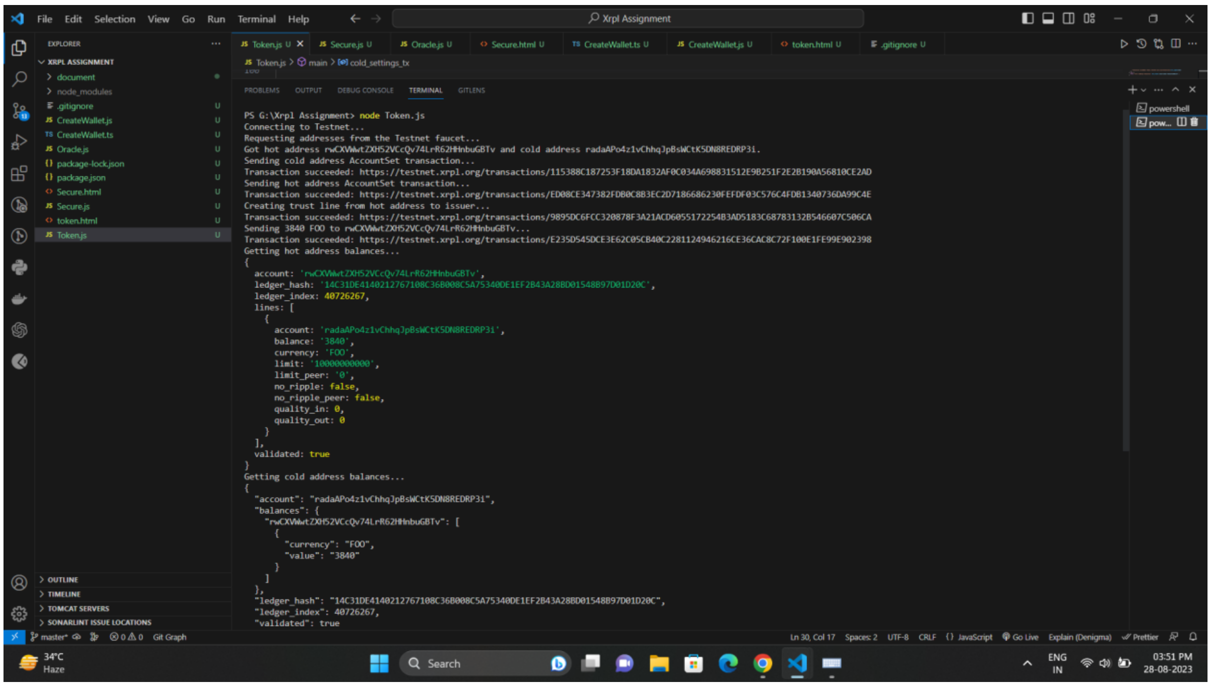


```javascript
  } else {
    throw `Error sending transaction: ${ts_result.result.meta.TransactionResult}`;
  }

  // Send token ----------------------------------------------------------------
  const issue_quantity = "3840";
  const send_token_tx = {
    TransactionType: "Payment",
    Account: cold_wallet.address,
    Amount: {
      currency: currency_code,
      value: issue_quantity,
      issuer: cold_wallet.address,
    },
    Destination: hot_wallet.address,
    DestinationTag: 1, // Needed since we enabled Require Destination Tags
    // on the hot account earlier.
  };

  const pay_prepared = await client.autofill(send_token_tx);
  const pay_signed = cold_wallet.sign(pay_prepared);
  console.log(
    `Sending ${issue_quantity} ${currency_code} to ${hot_wallet.address}...`
  );
  const pay_result = await client.submitAndWait(pay_signed.tx_blob);
  if (pay_result.result.meta.TransactionResult == "tesSUCCESS") {
    console.log(
      `Transaction succeeded: https://testnet.xrpl.org/transactions/${pay_signed.hash}`
    );
  } else {
    throw `Error sending transaction: ${pay_result.result.meta.TransactionResult}`;
  }

  // Check balances ------------------------------------------------------------
  console.log("Getting hot address balances...");
  const hot_balances = await client.request({
    command: "account_lines",
    account: hot_wallet.address,
    ledger_index: "validated",
  });
  console.log(hot_balances.result);

  console.log("Getting cold address balances...");
  const cold_balances = await client.request({
    command: "gateway_balances",
```



```javascript
    );
  } else {
    throw `Error sending transaction: ${pay_result.result.meta.TransactionResult}`;
  }

  // Check balances ------------------------------------------------------------
  console.log("Getting hot address balances...");
  const hot_balances = await client.request({
    command: "account_lines",
    account: hot_wallet.address,
    ledger_index: "validated",
  });
  console.log(hot_balances.result);

  console.log("Getting cold address balances...");
  const cold_balances = await client.request({
    command: "gateway_balances",
    account: cold_wallet.address,
    ledger_index: "validated",
    hotwallet: [hot_wallet.address],
  });
  console.log(JSON.stringify(cold_balances.result, null, 2));

  client.disconnect();
} // End of main()

main();
```

Output



```
PS G:\Xrpl Assignment> node Token.js
Connecting to Testnet...
Requesting addresses from the Testnet faucet...
Got hot address rwCXVWwtZXH52VCcQv74LrR62HHnbuGBTv and cold address radaAPo4z1vChhqJpBsWCtK5DN8REDRP3i.
Sending cold address AccountSet transaction...
Transaction succeeded: https://testnet.xrpl.org/transactions/115388C187253F18DA1832AF0C034A698831512E9B251F2E2B190A56810CE2AD
Sending hot address AccountSet transaction...
Transaction succeeded: https://testnet.xrpl.org/transactions/ED08CE347382FDB0C8B3EC2D7186686230FEFDF03C576C4FDB1340736DA99C4E
Creating trust line from hot address to issuer...
Transaction succeeded: https://testnet.xrpl.org/transactions/9895DC6FCC320878F3A21ACD6055172254B3AD5183C68783132B546607C506CA
Sending 3840 FOO to rwCXVWwtZXH52VCcQv74LrR62HHnbuGBTv...
Transaction succeeded: https://testnet.xrpl.org/transactions/E235D545DCE3E62C05CB40C2281124946216CE36CAC8C72F100E1FE99E902398
Getting hot address balances...
{
  account: 'rwCXVWwtZXH52VCcQv74LrR62HHnbuGBTv',
  ledger_hash: '14C31DE4140212767108C36B008C5A75340DE1EF2B43A28BD01548B97D01D20C',
  ledger_index: 40726267,
  lines: [
    {
      account: 'radaAPo4z1vChhqJpBsWCtK5DN8REDRP3i',
      balance: '3840',
      currency: 'FOO',
      limit: '10000000000',
      limit_peer: '0',
      no_ripple: false,
      no_ripple_peer: false,
      quality_in: 0,
      quality_out: 0
    }
  ],
  validated: true
}
Getting cold address balances...
{
  "account": "radaAPo4z1vChhqJpBsWCtK5DN8REDRP3i",
  "balances": {
    "rwCXVWwtZXH52VCcQv74LrR62HHnbuGBTv": [
      {
        "currency": "FOO",
        "value": "3840"
      }
    ]
  },
  "ledger_hash": "14C31DE4140212767108C36B008C5A75340DE1EF2B43A28BD01548B97D01D20C",
  "ledger_index": 40726267,
  "validated": true
```