Lesson 9 - Ethers / unit testing

This week

Mon : Ethers
Tues : Intro to DeFi
Weds : Decentralised storage
Thurs : Gas Optimisation

Ethers JS

See Documentation
and cheat sheet

Connecting to the network

1. Metamask

   Metamask will

   > - Act as a connection to the blockchain network, via an RPC node (a Provider)
   > - Sign transations (a Signer)

2. Use a JsonRpcProvider

Providers

A generic API for **account-less** blockchain interaction, regardless of backend

Ethers gives us some specific providers, for example to connect to localhost via the web socket provider

```
const provider = new
ethers.providers.WebSocketProvider("ws://127.0.0.1:8545");
```

or to connect to a generic JsonRpc endpoint

```
const provider = new ethers.provider.JsonRpcProvider(`rpc
endpoint url`);
```

Getting the Chain ID

Once we have setup the provider, we can get the network and from that the ChainID

```
const network = await provider.getNetwork();
const chainId = network.chainId;
```

Signers

Signers are an abstraction of accounts, it has concrete sub classes Wallet, VoidSigner or JsonRpcSigner.

Blockchain details from the signer

```
signer.getBalance( [ blockTag = "latest" ] ) ⇒ Promise<
BigNumber
```

Returns the balance of this wallet at blockTag.

```
signer.getChainId( ) ⇒ Promise< number >
```

Returns the chain ID this wallet is connected to.

```
signer.getGasPrice( ) ⇒ Promise< BigNumber >
```

Returns the current gas price.

```
signer.getTransactionCount([ blockTag = "latest" ]) ⇒
Promise<number> ```
Returns the number of transactions this account has ever sent.
This is the value required to be included in transactions as
the nonce.

```js
signer.call( transactionRequest ) ⇒ Promise< string<
DataHexString >
```

Returns the result of calling using the transactionRequest, with this account address being used as the from field.

```
signer.estimateGas( transactionRequest ) ⇒ Promise< BigNumber >
```

Returns the result of estimating the cost to send the transactionRequest, with this account address being used as the from field.
```

```
signer.resolveName( ensName ) ⇒ Promise< string< Address > >
```

Returns the address associated with the ensName.

Using a wallet

You can create a wallet in the following ways

```
new ethers.Wallet( privateKey [ , provider ] )
ethers.Wallet.createRandom( [ options = {} ] ) ⇒ Wallet
ethers.Wallet.fromEncryptedJson( json , password [ , progress ]
) ⇒ Promise< Wallet>
ethers.Wallet.fromEncryptedJsonSync( json , password ) ⇒ Wallet
ethers.Wallet.fromMnemonic( mnemonic [ , path , [ wordlist ] ]
) ⇒ Wallet
```

Once you have created a wallet, there are a number of methods you can use

```
wallet = walletMnemonic.connect(provider)
await walletMnemonic.getAddress()
await walletMnemonic.signMessage("Hello Bootcamp")
await wallet.sendTransaction(tx)
```

Interacting with contracts

Human Readable ABI

See docs

```
const humanReadableAbi = [
  "function transferFrom(address from, address to, uint
value)",
  "function balanceOf(address owner) view returns (uint
balance)",
  "event Transfer(address indexed from, address indexed to,
address value)"
  ];
```

A Human-Readable ABI is simple an array of strings, where each string is the Solidity signature.

Signatures may be minimally specified (i.e. names of inputs and outputs may be omitted) or fully specified (i.e. with all property names) and whitespace is ignored.

Several modifiers available in Solidity are dropped internally, as they are not required for the ABI and used only by Solidity's semantic checking system, such as input parameter data location like `"calldata"` and `"memory"`. As such, they can be safely dropped in the ABI as well.

Connecting to an existing contract

Requirements :

- Contract Address (or ENS name)
- Contract ABI
- Provider

```
const daiAddress = "dai.tokens.ethers.eth";


// You can specify the essential parts of the ABI in a human
readable format
const daiAbi = [

  "function name() view returns (string)",
  "function symbol() view returns (string)",
  "function balanceOf(address) view returns (uint)",
  "function transfer(address to, uint amount)",

  "event Transfer(address indexed from, address indexed to,
uint amount)"
];

// Once you have those parts you can create the contract object
const daiContract = new ethers.Contract(daiAddress, daiAbi,
provider);
```

This class is a *meta-class*, which means its methods are constructed at runtime, and when you pass in the ABI to the constructor it uses it to determine which methods to add.

Calling view functions

Give a contract object , you can call functions within that contract

```
await daiContract.name()
await daiContract.symbol()
```

Creating transactions

Sending an simple transaction

```
const signer = provider.getSigner();

const destination =
"0xc9ec550bea1c64d779124b23a26292cc223327b6";
const amount = ethers.utils.parseEther("2.0");

// Submit transaction to the blockchain
const tx = await signer.sendTransaction({
to: destination,
value: amount,
maxPriorityFeePerGas: "3000000000", // Max priority fee per gas
maxFeePerGas: "5000000000000", // Max fee per gas
});

// Wait for transaction to be mined
const receipt = await tx.wait();
```

Calling a state changing function

```
const contractWithSigner = contract.connect(signer);
const amount = ethers.utils.parseUnits("1.0", 18);
tx = contractWithSigner.transfer("vitalik.eth", amount);
```

Interacting with events

We can use the contracts API to work with events
See Documentation

Methods

query Filter

```
contract.queryFilter( event [ , fromBlockOrBlockHash [ ,
toBlock ] ) ⇒ Promise< Array< Event > >
```

Return Events that match the *event*.

listenerCount

```
contract.listenerCount( [ event ] ) ⇒ number
```

Return the number of listeners that are subscribed to *event*. If no event is provided, returns the total count of all events.

listeners

```
contract.listeners( event ) ⇒ Array< Listener >
```

Return a list of listeners that are subscribed to *event*.

off

```
contract.off( event , listener ) ⇒ this
```

Unsubscribe *listener* to *event*.

on

```
contract.on( event , listener ) ⇒ this
```

Subscribe to *event* calling *listener* when the event occurs.

once

```
contract.once( event , listener ) ⇒ this
```

Subscribe once to *event* calling *listener* when the event occurs.

removeAllListeners

```
contract.removeAllListeners( [ event ] ) ⇒ this
```

Unsubscribe all listeners for *event*. If no event is provided, all events are unsubscribed.

Examples

If you have set up the `contract` object, you could react to `Transfer` and `Mint` events as follows

```
contract.on("Transfer", (from, to, value) => {
    console.log(`${value} tokens were transfered from ${from}
to ${to}`);
});

contract.on("Mint", (tokenId) => {
    console.log(`Token with ID ${tokenId} has been minted`);
});
```

Utility functions

See Conversion

There are functions to convert between denominations

```
const oneGwei = BigNumber.from("1000000000"); const oneEther =
BigNumber.from("1000000000000000000");

formatUnits(oneGwei, 0);
// '1000000000'

formatUnits(oneGwei, "gwei");
// '1.0'

formatUnits(oneEther);
// '1.0'


parseUnits("121.0", "gwei");
// { BigNumber: "121000000000" }

parseEther("1.0");
// { BigNumber: "1000000000000000000" }
```

BNB Chain public RPC nodes

See Docs

Setting up the provider

Mainnet

```
const provider = new
ethers.provider.JsonRpcProvider(https://bsc-
dataseed1.binance.org/);
```

Testnet

```
const provider = new
ethers.provider.JsonRpcProvider(https://data-seed-prebsc-1-
s1.binance.org:8545/);
```

Unit Testing in Hardhat

Deploying to a network

scripts > JS deploy_script.js > ...

```
const hre = require("hardhat");

async function main() {
  // Hardhat always runs the compile task when running scripts with its command
  // line interface.
  //
  // If this script is run directly using `node` you may want to call compile
  // manually to make sure everything is compiled
  // await hre.run('compile');

  // We get the contract to deploy
  const [deployer] = await ethers.getSigners();
  console.log(`Deploying contracts with the account: ${deployer.address}`);

  const balance = await deployer.getBalance();
  console.log(`Account Balance: ${balance.toString()}`);

  // We get the contract to deploy
  const DummyContract = await hre.ethers.getContractFactory("DummyContract");
  const dummyContract = await DummyContract.deploy();

  console.log("Token deployed to:", dummyContract.address);
}

```

The steps are

1. Getting a reference to the contract using contractFactory

```
const DummyContract = await
ethers.getContractFactory(“DummyContract”);
```

2. Deploying the contract

```
const dummyContract = await DummyContract.deploy();
```

If you need a reference to the address (for example if you need it when deploying to another contract) you can use

```
dummyContract.address
```

Setting up accounts

we can use getSigners to assign accounts to variables, the available accounts will depend on what is supplied by the framework.

```
[owner, addr1, addr2, _] = await ethers.getSigners();
```

Signer

Signer is a generic API for creating trusted **account-based** data
The most common Signers you will encounter are:

1. Wallet, which is a class which knows its private key and can execute any operations with it
2. JsonRpcSigner, which is connected to a JsonRpcProvider (or sub-class) and is acquired using getSigner

Calling functions from different accounts

1. Get the signers object

   ```
   const [owner, addr1] = await ethers.getSigners();
   ```

2. Use the connect method to change the signer

```
        await greeter.connect(addr1).setGreeting("Hallo,
Erde!");
```

Writing unit tests

We go through a similar process of deployment when writing unit tests, but you need to decide when you will deploy the contracts

1. You can redeploy the contracts before each test
2. You can deploy the contracts once, then run a set of tests
   This decision will likely depend on the time taken for deployment.

Unit tests use the mocha and chai utilities

```
const { expect } = require("chai");
const { ethers } = require("hardhat");

describe("DummyContract", function () {
    // Initialise variables
    let DummyContract, dummyContract, owner, addr1, addr2;

    beforeEach(async () => {
        // Deploy a new instance of the contract
        DummyContract = await
ethers.getContractFactory("DummyContract");
        dummyContract = await DummyContract.deploy();
        // Get accounts and assign to pre-defined variables
        [owner, addr1, addr2, _] = await ethers.getSigners();
    });

    describe("Deployment", () => {
        it("Should be set with the Dummy Contract information",
async () => {
            // Failing test
            expect(addr1.address).to.not.equal(await
dummyContract.owner());

            // Passing tests
            expect(await
dummyContract.owner()).to.equal(owner.address);
            expect(await
dummyContract.symbol()).to.equal("DumTkn");
        });
    });

    describe("setUp", () => {
        it("Should not allow anyone but the owner to call",
async () => {
            await expect(() =>
                dummyContract
                    .connect(addr1)
                    .setUp()
                    .to.be.revertedWith("Ownable: caller is not
the owner")
            );
        });

        it("Should mint the initial amount to the contract
```

```
owner", async () => {
            const ownerBalanceBefore = await
dummyContract.balanceOf(owner.address);
            await dummyContract.setUp();
            const ownerBalanceAfter = await
dummyContract.balanceOf(owner.address);

expect(ownerBalanceAfter).to.equal(ownerBalanceBefore + 100);
        });
    });
});
```

Running the tests in hardhat

```
npx hardhat test
```

Will run the tests that are available in the tests folder.

Useful Chai matchers

See docs
Include chai with

```
const { expect } = require("chai");
```

1. Test BigNumbers

```
        expect(await
token.balanceOf(wallet.address)).to.equal(993);
        expect(BigNumber.from(100)).to.be.within(BigNumber.from
(99), BigNumber.from(101));
expect(BigNumber.from(100)).to.be.closeTo(BigNumber.from(101),
10);
        ```

2. Emitting events
```js
await expect(token.transfer(walletTo.address, 7))
  .to.emit(token, 'Transfer')
  .withArgs(wallet.address, walletTo.address, 7);
```

We can match on indexed events

3. Called on contract (with arguments)

```
        await token.balanceOf(wallet.address)

        expect('balanceOf').to.be.calledOnContractWith(token,
[wallet.address]);
```

4. Revert (with message)

```
await expect(token.transfer(walletTo.address, 1007))
  .to.be.revertedWith('Insufficient funds');
```

You can use regular expressions

```
await expect(token.checkRole('ADMIN'))
  .to.be.revertedWith(/AccessControl: account .* is missing
role .*/);
```

5. Change ether balance

```
        await expect(() => wallet.sendTransaction({to:
walletTo.address, value: 200}))
  .to.changeEtherBalance(walletTo, 200);

await expect(await wallet.sendTransaction({to:
walletTo.address, value: 200}))
  .to.changeEtherBalance(walletTo, 200);
        ```
```
```