Lesson 9 - Ethers / unit testing

This week

Mon : Ethers
Tues : Team security game
Weds : Decentralised storage
Thurs : Gas Optimisation

Ethers

Providers

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

getDefaultProvider() choices randomly 2/5 nodes.

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

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

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 old 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.

Creating contract instances

See docs
Use

```
new ethers.Contract( address , abi , signerOrProvider )
```

By passing in a Signer. this will return a **Contract** which will act on behalf of that signer.

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

A generic API for creating trusted(i.e. signed) **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.name](http://dummycontract.name/)
()).to.equal(“DummyToken”);
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

```
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
   `Javascript= 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

```
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 even 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);
```

---

### Making a fork of Mainnet

See hardhat documentation

You first need to have an account on Infura or Alchemy
This will give you a key so that you can use their RPC nodes.

#### Forking using ganache

`npx ganache-cli --f https://mainnet.infura.io/v3/<your key> -m "your 12 word mnemonic" --unlock <address> -i <chain ID>`

#### Fork from hardhat

```
npx hardhat node --fork https://eth-
mainnet.alchemyapi.io/v2/<your key>
```

In hardhat you can also specify this in the config file

```
networks: {
  hardhat: {
    forking: {
      url: "https://eth-mainnet.alchemyapi.io/v2/<key>",
    }
  }
}
```

If you want to pin to a certain blocknumber you can also add that as a parameter
```

```
npx hardhat node --fork https://eth-
mainnet.alchemyapi.io/v2/<key> --fork-block-number 11095000
```

Token Standards

ERC20

ERC20 is an example of a fungible token
It is a standard accepted by developers, exchanges and wallet creators
To be an ERC20 token, your contract must implement the following functions and events

```
FUNCTIONS

totalSupply()

balanceOf(account)

transfer(recipient, amount)

allowance(owner, spender)

approve(spender, amount)

transferFrom(sender, recipient, amount)
```

```
EVENTS

Transfer(from, to, value)

Approval(owner, spender, value)
```

ERC721

ERC721 is an example of a non fungible token (NFT)
It has the following functions and events

```
FUNCTIONS

balanceOf(owner)

ownerOf(tokenId)

safeTransferFrom(from, to, tokenId)

transferFrom(from, to, tokenId)

approve(to, tokenId)

getApproved(tokenId)

setApprovalForAll(operator, _approved)

isApprovedForAll(owner, operator)

safeTransferFrom(from, to, tokenId, data)
```

```
EVENTS

Transfer(from, to, tokenId)

Approval(owner, approved, tokenId)

ApprovalForAll(owner, operator, approved)
```

Useful Open Source Collections

About Open Zeppelin:

Open Zeppelin are well known in the Ethereum community. They provide a set of audited smart contracts and libraries that are a standard in the industry.

Inheriting these contracts will provide a significantly higher degree of security and robustness in your code.

See https://docs.openzeppelin.com/contracts/4.x/

Open Zeppelin Token Contracts

Even though the concept of a token is simple, they have a variety of complexities in the implementation. Because everything in Ethereum is just a smart contract, and there are no rules about what smart contracts have to do, the community has developed a variety of standards (called EIPs or ERCs) for documenting how a contract can interoperate with other contracts.

- ERC20: the most widespread token standard for fungible assets, albeit somewhat limited by its simplicity.

- ERC721: the de-facto solution for non-fungible tokens, often used for collectibles and games.

- ERC777: a richer standard for fungible tokens, enabling new use cases and building on past learnings. Backwards compatible with ERC20.

- ERC1155: a novel standard for multi-tokens, allowing for a single contract to represent multiple fungible and non-fungible tokens, along with batched operations for increased gas efficiency.

Safe Functions

Safe functions (SafeTransferFrom etc.) were introduced to prevent tokens being sent to contracts that didn't know how to handle them, and thus becoming stuck in the contract.

Open Zeppelin Access Control / Security Contracts

- Ownable
- AccessControl
- TimeLockController
- Pausable
- Reentrancy Guard
- PullPayment

Open Zeppelin Governance Contracts

Implements on-chain voting protocols similar to Compound’s Governor Alpha & Bravo

Open Zeppelin Cryptography Contracts

- ECDSA contract for checking signatures.
- MerkleProof for proving an item is in a Merkle tree.

Open Zeppelin Introspection Contracts

Contracts to allow runtime checks whether a target contract implements a particular interface

Open Zeppelin Maths Contracts

SafeMath - to prevent under / overflow etc. Some of this functionality is part of Solidity since version 0.8.0

Open Zeppelin Payment Contracts

- Payment splitter
- Escrow

Open Zeppelin Collections Contracts

- Enumerable Set
- Enumerable Map

Open Zeppelin Miscellaneous Contracts

- Address
- Multicall

Open Zeppelin Upgradability Contracts

- Proxy

Importing from Github in Remix

See Documentation
You can import directly from github or npm

import "https://github.com/OpenZeppelin/openzeppelin-contracts/contracts/access/Ownable.sol";

or

```
import "@openzeppelin/contracts@4.2.0/token/ERC20/ERC20.sol";
```

References

Solidity Documentation
Libraries