# Connecting Accounts and Calling Functions

When working with smart contracts, you might want to call functions from different accounts. Here's how you can do it:

- Connecting Accounts:
    - When creating a smart contract.
    - When writing a transfer, using the `accounts` object.

```
// Example of calling a function with a specific account
contract.connect(anotherAccount).functionName(parameters);
```

- VM Syntax: When calling functions with parameters in VM, pass all parameters in an array.

## Scripts vs. Tests

This week's project involves using **scripts** instead of tests to interact with the smart contract.

- Transitioning from Tests to Scripts:
    1. Work in the tests first, debug, and use `console.log` to understand how everything works.
    2. Once you understand the logic, copy the code over to a script.
    3. Execute the script.
- Weekend Project: Develop and run scripts for `ballot.sol` to interact with it within your group.
    - Write a script to **deploy** the smart contract.
    - Write a script to **give right to vote** for people.
    - Write a script to **cast a vote** using your account.

## Project Setup

When setting up the project, be cautious about copying unnecessary files.

- **Copying Project Folders**:
  - Copy necessary files from `project3` to `project4`.
  - Ensure hidden files and folders are copied.
- **Files to Avoid**:
  - **`node_modules`**: This folder is large and can be reinstalled.
  - `artifacts/cache`: Avoid copying to save time.

  The `node_modules` folder contains all the installed dependencies for your project. It's usually quite large, so it's best to avoid copying it and reinstall the dependencies instead.

- **Installation**: Run `npm install` (or `npm i`) to install dependencies.
- **Cleaning**: Use `npx hardhat clean` to start in a new state.

## Avoiding Old Models

It is important to avoid old models. To ensure everything is working correctly, compile using the command:

```
npx hardhat compile
```

To run tests and check the current project status, use:

```
npx hardhat test
```

This command will show the testing situation, indicating what passes and what is not yet implemented.

## Hardhat Installation and Usage

### Global vs. Local Installation

Avoid installing Hardhat globally. It is designed to be installed within each project folder. If you encounter an issue where Hardhat tries to install globally, ensure you are inside your project folder before running commands like `npx hardhat clean`.

If you accidentally install Hardhat globally:

1. Uninstall it globally.
2. Ensure your project folder contains a `package.json` file.
3. Run Hardhat commands from within the project folder.

### Project Setup

To set up your project:

1. Open your project folder in VS Code or your preferred file explorer.

2. Ensure you have a populated `.env` file. This file should contain:

    - A `PRIVATE_KEY` or `MNEMONIC` for your Ethereum account. Using a private key is preferred to avoid exposing your entire mnemonic.
    - At least one API key from services like Alchemy, Etherscan, or Infura.

### Installing Node Modules

To install the necessary node modules, navigate to your project folder in the terminal and run:

```
npm install
```

This command is generally faster than copying node modules from other directories.

## Deploying Smart Contracts

### Skipping Ignition

Hardhat suggests using an ignition folder for deploying and managing smart contracts. This involves tracking past deployments and changes. However, this method can be too opinionated. Instead, you can write your own scripts for deploying smart contracts and managing blockchain interactions.

## Creating Deployment Scripts

1. Create a folder named `scripts` in your project.
2. Inside the `scripts` folder, create a TypeScript file, such as `deploy.ts`.

   ```
   console.log('deploy');
   ```

   This script can contain any TypeScript syntax and utilize the Hardhat runtime environment.

## Running Scripts

To execute the script, use the following command:

```
npx hardhat run scripts/deploy.ts
```

This command runs the script directly.

## Testing Scripts

To test scripts, use:

```
npx hardhat test
```

# Key Differences: Run vs. Test

| Feature | Run | Test |
|---|---|---|
| Purpose | To perform a specific action, like deploying a contract. | To run a batch of operations in sequence with care and an emulated virtual environment. |
| Usage | Executing a script for real actions. | Testing multiple scenarios in parallel within a controlled environment. |
| Output | Basic output from the script. | Detailed information related to testing, including test results and debugging information. |
| Suite | Not part of a test suite. | Part of a test suite, allowing multiple tests to run in parallel. |
| Analogy | Like a real football game where actions are recorded. | Like practicing with club players in-house, where you can repeat operations many times. |
| Command Example | `npx hardhat run scripts/deploy.ts` | `npx hardhat test` |
| Structure | No `describe` blocks. | Uses `describe` blocks to define test scenarios. |
| Environment | A real environment. | An emulated environment. |
| Use Case | Deploying a smart contract to a blockchain. | Running various tests in parallel. |
| Operation | Is a real thing. | Is a virtual environment to test operations. |
| Final | Actions are recorded. | Can be repeated hundreds of times. |
| Result | Transactions are for real. | Doesn't guarantee real-world reliability. |
| Note | Everything you do is for real and can get you in trouble. | The things that were done testing may not be the same as when they were in production. |

Test is the way to run a batch of things in sequence, and the script is what you want to do something with, like sending a transaction or deploying a smart contract.

## Understanding Scripts in Hardhat

## Declaring and Calling Functions

The lecture discusses defining and calling functions in a script using TypeScript. It emphasizes that merely defining a function is insufficient; it must be called to execute.

- Function Declaration: This is where you define what a function does.
- Function Call: This is where you actually execute the function.

Analogy:

> The professor uses the analogy of a wife asking her husband to buy milk, oil, and bread. Simply stating the need doesn't fulfill it; the husband needs to go to the supermarket and buy those items. Similarly, defining a function doesn't execute it until it's called.

Here's a simple example of declaring and calling a function in TypeScript:

```
function exampleFunction() {
  console.log("Function executed!");
}

exampleFunction(); // Calling the function
```

## Reusing Tests in Scripts

A key point is the reusability of tests in scripts. Code from tests can be copied and pasted into scripts to perform actions like deploying contracts.

- Copy-Pasting: Code snippets from tests can be used in scripts.
- Modification: Once pasted, the code can be modified to suit the script's specific needs.

```
// Example: Copying code from tests to deploy a contract in a script
// Test code (Hypothetical)
const walletClient = await getWalletClient();
const ballotContract = await deployBallotContract();

// Script code
console.log("Deploying the ballot...");
const ballotContractAddress = await deployBallotContract();
console.log(`Ballot contract deployed at ${ballotContractAddress}`);
```

## Step-by-Step Scripting Process

1. Copy Initial Code: Start by copying the first few lines of code that set up the environment.
2. Integrate Additional Code: Add necessary functions or operations (e.g., deploying a contract).
3. Logging: Use `console.log` to print relevant information, helping you track the script's progress and the values of variables.

```
// Example: Deploying a ballot contract and logging proposals
// Copy these initial lines
import { ethers } from "hardhat";

async function main() {
  // Add the code to deploy the contract and fetch proposals
  console.log("Deploying the ballot...");
  const ballotContract = await deployBallotContract();
  const proposals = await ballotContract.getProposals();
  console.log("Proposals:", proposals);
}

main().catch((error) => {
  console.error(error);
  process.exitCode = 1;
});
```

## Running Scripts in Different Environments

The lecture touches on running scripts in different environments, from the Hardhat local environment to public testnets.

| Environment | Description |
|---|---|
| Hardhat Network | Local development environment, useful for testing without real funds. |
| Public Testnets | Public blockchain networks used for testing smart contracts before deploying them to the mainnet. Examples: Sepolia, Goerli. |
| Mainnet | The main Ethereum network where real transactions occur. |

### Connecting to Public Blockchains

Connecting to a public blockchain involves using an RPC (Remote Procedure Call) provider. Chainlist is mentioned as a resource for finding RPC providers for various blockchains and testnets.

## Connecting to Public Blockchains with RPC Nodes

### Understanding RPC Nodes

- RPC nodes, or Remote Procedure Call nodes, allow connection to blockchains without running your own blockchain instance.
- They provide access to:
    - Blocks
    - Transactions
    - Broadcasting transactions

  RPC nodes act as a middleman, handling transactions without the ability to alter them.

### Public RPC Providers

- Public RPC providers offer free access to blockchains up to certain rate limits.
- If the number of calls exceeds the limit, the provider might block access.

### Configuring Hardhat for Testnets

- Modify the `hardhat.config.ts` file to include configurations for test networks like Sepolia.
- Add network configurations with the URL of a public node. Example configuration:

```
module.exports = {
  networks: {
    sepolia: {
      url: "YOUR_SEPOLIA_RPC_URL",
      chainId: 11155111
    }
  }
};
```

- Use the `--network` flag to specify the network when running deployment or other scripts.

## Interacting with Testnets in Scripts

### Using `publicClient`

- Import `publicClient` to interact with blockchain data in your scripts.
- Example of importing public client:

```
import { publicClient } from '@wagmi/core';
```

- Utilize `publicClient.getBlock()` to retrieve the latest block or a specific block by number.
- Example usage:

```
const lastBlock = await publicClient.getBlock();
console.log(lastBlock);
```

### Demonstrating Network Targeting

- Running scripts without the `--network` tag connects to the Hardhat runtime environment.
- Using the `--network sepolia` flag retrieves real blockchain data from the Sepolia testnet.
- This demonstrates the ability to target different networks with a single flag.

## Setting Up Environment Variables

### Environment File

- Create a `.env` file in the root of your project to store sensitive information such as private keys.

### Security Considerations

- Never store mainnet private keys in your development environment.
- Use a new wallet specifically for development purposes to mitigate risks.
- Ensure that the testnet account and mainnet account are not linked to the same private key to avoid potential security breaches.

## Configuring Environment Variables

To manage environment variables such as API keys and private keys, you can use the `INF` package. This allows you to securely access these variables in your scripts.

1. Install the `INF` package using the following command in your terminal, within the same directory as your `package.json` file:

   ```
   npm install INF
   ```

2. Access the environment variables in your Hardhat configuration (`hardhat.config.js`) or directly within your scripts.

   - Example of accessing environment variables:

     ```
     const providerApiKey = process.env.PROVIDER_API_KEY;
     const deployerPrivateKey = process.env.DEPLOYER_PRIVATE_KEY;
     ```

3. To verify that your environment variables are correctly configured, you can use `console.log` to display the first few digits of your API key and private key:

   ```
   console.log("API Key (first 5 digits):", providerApiKey.slice(0, 5));
   console.log("Private Key (first 5 digits):", deployerPrivateKey.slice
   ```

   *It's important to only show a substring of your keys to avoid exposing sensitive information while streaming or sharing your screen.*

4. If the console output is empty, double-check your setup and ensure that the variables are correctly set in your `.env` file.

## Setting Up Accounts in Hardhat Config

To configure your wallet for deploying smart contracts, you need to set up your accounts in `hardhat.config.js`. This involves specifying the URL to connect to a network (e.g., Goerli) and providing your private key.

1. Update your `hardhat.config.js` file to include the network configuration:

   ```
   module.exports = {
     networks: {
       goerli: {
         url: "YOUR_ALCHEMY_GOERLI_URL", // Replace with your Alchemy o
         accounts: ["YOUR_PRIVATE_KEY"] // Replace with your private ke
       }
     }
   };
   ```

   - Replace `"YOUR_ALCHEMY_GOERLI_URL"` with the URL provided by your RPC provider (e.g., Alchemy, Infura).
   - Replace `"YOUR_PRIVATE_KEY"` with your account's private key.

2. By setting up your private key with a provider, you gain better rate limits and the ability to send transactions without being throttled.

3. Hardhat uses the first account in the `accounts` array as the default account for signing transactions. You can include multiple private keys if you have multiple accounts.

## Deploying Smart Contracts and Viewing Blockchain Information

Before deploying a smart contract, it's helpful to view blockchain information such as the block number, deployer address, and account balance. This can be done by prepending code to your deployment script.

1. Add the following code snippet to your deployment script:

```
async function main() {
  const [deployer] = await ethers.getSigners();

  console.log("Deploying contracts with the account:", deployer.addre

  const balance = await deployer.getBalance();
  console.log("Account balance:", balance.toString());

  const Contract = await ethers.getContractFactory("YourContract");
  const contract = await Contract.deploy();
  await contract.deployed();

  console.log("Contract deployed to address:", contract.address);
}
```

   - Ensure you replace `"YourContract"` with the name of your contract.

2. When you run your deployment script, this code will display:

   - The address of the deployer account
   - The account balance
   - The address to which the contract is deployed.

3. It's crucial to understand that your private key is used to **sign transactions locally** and is **never exposed** to public nodes. The signed transactions are then sent to the public nodes for execution.

## Security Best Practices

When dealing with private keys and API keys, always follow these security practices:

- **Never expose your private key.** It should never leave your device.
- When sharing your screen or streaming, be cautious about displaying sensitive information. Use methods like `slice` to show only a portion of the key for verification purposes.
- Treat testnet funds and keys with the same level of caution as real funds to build good security habits.

  Private Key Definition

A **private key** is a secret, cryptographic key that allows you to control the funds in a blockchain account. Anyone who has access to your private key has access to your funds, so it is essential to keep it secure.

## Deploying Smart Contracts to a Public Testnet

To deploy a smart contract to a public testnet (e.g., Sepolia), you need to:

- Pass the **URL configuration**.
- Configure the **accounts**.
- Use the **network flag** (e.g., `--network sepolia`).

This tells Hardhat which configurations to use, which account to use, and deploys the smart contract to the specified public testnet.

## Provider API Key and Deployer Private Key

- Store the **Provider API Key** and **Deployer Private Key** in a `.env` file.
- Do not hardcode these keys directly in the script.

## Limitations of Hardhat

Hardhat uses **Ethereum Virtual Machine (EVM)** to communicate with the blockchain. While Hardhat is useful, it has limitations when building complex applications. We may need to bypass Hardhat and use **VM directly**.

## Deploying with VM Directly

- Create a new script called `deploy_with_vm.ts`.
- Hardhat is not designed to take arguments directly from the script. For example, if you wanted to dynamically assign votes to people, you could not pass these arguments directly through Hardhat because Hardhat does not recognize them.
- You must use **TS Node** to deploy directly with VM to get around this limitation.

## Process Arguments (argv)

- Use `process.argv` from Node.js to pass arguments to the script.
- `process.argv` allows you to pick all the values passed as arguments from the command line.

```
// Example: Accessing command line arguments using process.argv
console.log(process.argv);
```

## Slicing Arguments

When using `process.argv`, the first two elements are typically:

1. The binary of TS Node being used to execute the script.
2. The script being executed.

Therefore, to access the actual arguments passed, you need to slice the array starting from index 2.

```
const args = process.argv.slice(2);
console.log(args);
```

## Running Scripts with TS Node

To execute a script with arguments, use TS Node directly.

```
ts-node filename.ts arg1 arg2 arg3
```

This command uses TS Node to execute the script and passes the specified arguments.

## Example Scenario: Passing Proposals

Suppose you want to pass proposals dynamically to a script. You can use `process.argv` to capture these proposals.

```
// deploy_with_vm.ts

async function main() {
    const proposals = process.argv.slice(2);

    if (proposals.length === 0) {
        console.error("Error: Proposals not provided.");
        return;
    }

    console.log("Proposals:", proposals);
}

main().catch((error) => {
    console.error(error);
    process.exitCode = 1;
});
```

To run this script with TS Node:

```
ts-node deploy_with_vm.ts proposal1 proposal2 proposal3
```

This will output:

```
Proposals: [ 'proposal1', 'proposal2', 'proposal3' ]
```

## Replacing Hardhat with Basic Viem Functions

Now that you can pass arguments, you can replace Hardhat helper functions with basic Viem functions. This includes using public providers and wallet clients directly.

## Running Scripts with VM Directly

We can run scripts on-chain using VM (likely referring to a Virtual Machine in the context of blockchain development) directly. This involves creating scripts for blockchain operations such as deploying smart contracts and dynamically granting voting rights.

## Creating Public Clients with VM

- Instead of relying on Hardhat, we can use VM directly to create public clients.
- This involves a more manual configuration process.

### Steps:

1. Create a public client using VM.
2. Configure the chain.
3. Create a transport object with the HTTP string, including the provider API key. This replaces the `hardhat.config.js` file's role.
4. Retrieve information such as the block number.

## Example

```
// Customize line 14 with your provider API key
// e.g., Alchemy, Infura
```

## Benefits of Using VM Directly

- Direct Connection: Connects directly to the blockchain without the Hardhat tooling overhead.
- Lightweight: Avoids the heavy tooling associated with Hardhat, which can be cumbersome for simple tasks.
- Customization: Provides greater control over configuration.

## Creating Wallet Clients with VM

Creating wallet clients with VM is more complex.

### Steps:

1. Import necessary modules:
   - `createWalletClient`
   - `privateKeyToAccount` from VM
2. Convert the private key to an account.
3. Set the deployer private key in the script.
4. Configure the wallet client:
   - Specify the account.
   - Specify the chain.
   - Specify the transport (including the URL).

## Comparison

| Feature | Hardhat (with Ethers) | VM Directly |
|---|---|---|
| Configuration | Simpler | More Complex |
| Tooling | Heavy | Lightweight |
| Private Key Usage | Under the hood | Manual configuration |

## Deploying Smart Contracts with VM

We can deploy smart contracts by using the ABI (Application Binary Interface) and bytecode.

### Steps:

1. Import the ABI and bytecode from a JSON file (artifacts).
2. Use the ABI and bytecode to deploy the smart contract.

### Example:

```
//Pick ABI and bytecode from JSON
//Use ABI and bytecode to deploy ballot contract
```

## Deploying Smart Contracts with Viem

When deploying a smart contract using Viem, you need to provide both the ABI (Application Binary Interface) and the bytecode. This is in contrast to using Hardhat, where you only need to specify the artifact name and constructor arguments. Getting this information requires extracting it from the artifacts and passing it directly to Viem.

### Getting Public Information

To fetch public information, like proposals, from a deployed contract, you need the contract's address.

```
// Example of fetching proposals from a contract address using Viem
// Note: This is a conceptual example and might require adjustments based

import { createPublicClient, http } from 'viem';
import { mainnet } from 'viem/chains';

const client = createPublicClient({
  chain: mainnet,
  transport: http()
})

// const proposals = await client.readContract({
//   address: '0xContractAddress...',
//   abi: contractABI,
//   functionName: 'getProposals',
// })

// console.log(proposals);
```

You'll need to import `Hex` and `Address` from `viem`. This allows you to interact with pre-deployed contracts similar to how Hardhat operates, but with more explicit control over the parameters.

# Operation Scripts as Homework

As a weekend project, you are assigned to create operation scripts for a pre-deployed ballot contract. These scripts should include functionalities such as:

- Casting votes
- Giving right to vote
- Delegating votes

Each script will take parameters such as the contract address, the voter's address, and the proposal index.

## Script Details

- Cast Votes: Receives the contract address of a previously deployed ballot, deploys it with one script initially, and then builds a script to cast votes.
- Give Right to Vote: A script to grant voting rights to a specified address.
- Delegate: A script that allows one voter to delegate their voting power to another.

## Implementation Details

These operations should be implemented as TypeScript scripts and stored in a GitHub repository. The project requires the use of Viem to interact with the blockchain.

## Deliverables

1. GitHub Repository: Containing all the TypeScript scripts for the specified operations.
2. Report: Detailing the execution of each smart contract call, including transaction hashes from a public testnet.

## Passing Parameters

To pass parameters to the scripts, you'll need to use Viem. This exercise is designed to provide hands-on experience beyond Remix.