Trading Bot Demo

Technology Stack & Tools

- Solidity (Writing Smart Contract)
- Javascript (React & Testing)
- [Hardhat](https://hardhat.org/) (Development Framework)
- [Ethers.js](https://docs.ethers.io/v5/) (Blockchain Interaction)
- [Alchemy](https://www.alchemy.com/) (Blockchain Connection)
- [Balancer](https://balancer.fi/) (Flash Loan Provider)
- [Uniswap V3](https://docs.uniswap.org/contracts/v3/overview) (Exchange)
- [Pancakeswap V3](https://docs.pancakeswap.finance/) (Exchange)

Requirements For Initial Setup
- Install [NodeJS](https://nodejs.org/en/). We recommend using the latest LTS (Long-Term-Support) version, and preferably installing NodeJS via [NVM](https://github.com/nvm-sh/nvm#intro).
- Create an [Alchemy](https://www.alchemy.com/) account, you'll need to create an app for the Ethereum chain, on the mainnet network

Setting Up
1. Clone/Download the Repository

2. Install Dependencies:
`npm install`

3. Create and Setup .env
Before running any scripts, you'll want to create a .env file with the following values (see .env.example):

- **ALCHEMY_API_KEY=""**
- **PRIVATE_KEY=""** (Private key of the account to recieve profit/execute arbitrage contract)

4. Start Hardhat Node:
In your terminal run:
`npx hardhat node`

Once you've started the hardhat node, copy the private key of the first account as you'll need to paste it in your .env file in the next step.

*As a reminder, do **NOT** use or fund the accounts/keys provided by the hardhat node in a real production setting, they are to be only used in your local testing!*

5. Add Private Key to .env
Copy the private key of the first account provided from the hardhat node, and paste in the value for the **PRIVATE_KEY** variable in your .env file

6. Deploy Smart Contract
In a separate terminal run:
`npx hardhat run scripts/deploy.js --network localhost`

Sometimes the deployed address may be different when testing, and therefore you'll need to update the **ARBITRAGE_ADDRESS** inside of the *config.json*

7. Start the Bot
`node bot.js`

8. Manipulate Price
In another terminal run:
`npx hardhat run scripts/manipulate.js --network localhost`

About config.json
PROJECT_SETTINGS
Inside the *config.json* file, under the **PROJECT_SETTINGS** object, there are 2 keys that hold a boolean value:
- **isLocal**
- **isDeployed**

Both options depend on how you wish to test the bot. By default both values are set to true. If you set isLocal to false, and then run the bot this will allow the bot to monitor swap events on the actual mainnet, instead of locally.

isDeployed's value can be set on whether you wish for the abritrage contract to be called if a potential trade is found. By default isDeployed is
set to true for local testing. Ideally this is helpful if you want to monitor swaps on mainnet and you don't have a contract deployed.
This will allow you to still experiment with finding potential abitrage opportunites.

Some other variables under **PROJECT_SETTINGS** are:
- **ARBITRAGE_ADDRESS** (Address of the Arbitrage contract)
- **PRICE_UNITS** (How many decimals to display when logging price)
- **PRICE_DIFFERENCE** (Minimum price difference required for continuing execution, if lower, just continue monitoring for new swaps)
- **GAS_LIMIT** (Gas limit estimate for estimating gas fee. Not actually used in submitting a transaction)
- **GAS_PRICE** (Hardcoded gas price for estimating gas fee. Not actually used in submitting a transaction)

TOKENS
If you are looking to test different ERC20 tokens and pools, you'll want to update the TOKENS object, specifically:
- **ARB_FOR** (Address of the token you are arbitraging for)
- **ARB_AGAINST** (Address of the token you are arbitraging against)
- **POOL_FEE**

UNISWAP & PANCAKESWAP
For the example exchanges used, there are generally 3 contracts you'll want to interact with for a Uniswap V3 like exchange:
- **QUOTER_V3** (QuoterV2 Contract address for simulating swaps and estimating input/output)

- **FACTORY_V3** (UniswapV3Factory & PancakeV3Factory Contract for finding token pools)
- **ROUTER_V3** (SwapRouter Contract for performing actual swaps)

Note that for Uniswap V3 exchanges, they may have a Quoter and a QuoterV2 contract. The bot by default uses QuoterV2.
Note that for Uniswap V3 exchanges, they may have a SwapRouter and a SwapRouter02 contract. The bot by default uses SwapRouter.

Be sure to look at [Uniswap documentation](https://docs.uniswap.org/contracts/v3/overview) for learning more about their V3 exchanges.

Testing Bot on Mainnet
For monitoring prices and detecting potential arbitrage opportunities, you do not need to deploy the contract.

1. Edit config.json
Inside the *config.json* file, set **isDeployed** to **false** and **isLocal** to **false**.

2. Create and Setup .env
See step #4 in **Setting Up**

3. Run the bot
`node bot.js`

Keep in mind you'll need to wait for an actual swap event to be triggered before it checks the price.

Anatomy of bot.js
The bot is essentially composed of 5 functions.
- *main()*
- *checkPrice()*
- *determineDirection()*
- *determineProfitability()*
- *executeTrade()*

The *main()* function monitors swap events from both Uniswap V3 & Pancakeswap V3.

When a swap event occurs, the *eventHandler()* will be called. Inside of the *eventHandler()* it calls *checkPrice()*, this function will log the current price of the assets on both Uniswap & Pancakeswap, and return the `priceDifference`

Then *determineDirection()* is called, this will determine the direction of the trades, where to buy first, then where to sell. This function will return an array called `exchangePath` in *main()*. The array contains Uniswap & Pancakeswap objects that were created in *initialization.js*. If no array is returned, this means the `priceDifference` returned earlier is not higher than `difference`

If `exchangePath` is not null, then execution moves into *determineProfitability()*. This is where you can set some of your conditions on whether there is a potential arbitrage or not. This function returns either true or false.

If true is returned from *determineProfitability()*, then it calls *executeTrade()* where it makes the call to the arbitrage contract to perform the trade. Afterwards a report is logged, and the bot resumes to monitoring for swap events.

Modifying & Testing the Scripts
Both the *manipulate.js* and *bot.js* has been setup to easily make some modifications easy. Before the main() function in *manipulate.js*, there will be a comment: **// -- CONFIGURE VALUES HERE -- //**. Below that will be some constants you'll be able to modify such as the unlocked account, and the amount of tokens you'll want that account to spent in order to manipulate price (You'll need to adjust this if you are looking to test different pairs).

For *bot.js*, you'd want to take a look at the function near line 132 called *determineProfitability()*. Inside this function you can set your conditions and do your calculations to determine whether you may have a potential profitable trade on your hands. The idea is that the function is to return an object with 2 keys:

- `isProfitable` (Boolean)
- `amount` (BigInt)

`isProfitable` should be set to **true** if a profitable trade is possible, and **false** if not.

Note if you are doing an arbitrage for a different ERC20 token than the one in the provided example (WETH), then you may also need to adjust profitability reporting in the *executeTrade()* function.

Keep in mind, after running the scripts, specifically *manipulate.js*, you may need to restart your hardhat node, and re-deploy contracts to properly retest.

Additional Information
The *bot.js* script uses helper functions for fetching token pool addresses, calculating price of assets, and fetching liqudity. These functions can be found in the *helper.js* file inside of the helper folder.

The helper folder also has *server.js* which is responsible for spinning up a local server, and *initialization.js* which is responsible for setting up the blockchain connection, configuring Uniswap/Pancakeswap contracts, etc.

As you customize parts of the script it's best to refer to [Uniswap documentation](https://docs.uniswap.org/contracts/v3/overview) for a more detail rundown on the protocol and interacting with the V3 exchange.

Using other EVM chains
If you are looking to test on an EVM compatible chain, you can follow these steps:

1. Update config.json TOKENS fields

- **ARB_FOR=""**
- **ARB_AGAINST=""**
- **POOL_FEE=""**

Token addresses will be different on different chains, you'll want to reference blockchain explorers such as [Polyscan](https://polygonscan.com/) for Polygon for token addresses you want to test.

2. Update config.json exchange addresses

- **QUOTER_V3=""**
- **FACTORY_V3=""**
- **ROUTER_V3=""**

You'll want to update the quoter, router and factory addresses inside of the *config.json* file with the V3 exchanges you want to use. Based on the exchange you want to use, refer to the documentation for it's address.

3. Change RPC URL

Inside of *initialization.js*, you'll want to update the websocket RPC URL. Example of Polygon:
```
provider = new ethers.WebSocketProvider(`wss://polygon-mainnet.g.alchemy.com/v2/${process.env.ALCHEMY_API_KEY}`)
```

Inside of *hardhat.config.js*, you'll want to update the forking URL. Example of Polygon:
```
url: `https://polygon-mainnet.g.alchemy.com/v2/${process.env.ALCHEMY_API_KEY}`,
```

4. Changing Arbitrage.sol
You may also need to change the flashloan provider used in the contract to one that is available on your chain of choice. Currently Balancer seems to support the following chains:
- Ethereum Mainnet
- Arbitrum
- Optimism
- Polygon
- Gnosis
- Avalanche
- Goerli (Testnet)
- Sepolia (Testnet)

Be sure to check their documentation for latest updates regarding their contracts and deployment addresses:
- [Balancer Documentation](https://docs.balancer.fi/)
- [Balancer Flash Loans](https://docs.balancer.fi/guides/arbitrageurs/flash-loans.html)

Additional Notes

- If testing out the *manipulate.js* script, you'll also want to update the **UNLOCKED_ACCOUNT** variable and adjust **AMOUNT** as needed.