Course
Fog Computing for Smart Services

Development guide for Smart Contracts

University of Ljubljana
Faculty of Computer and Information Science

DESIGN PATTERNS

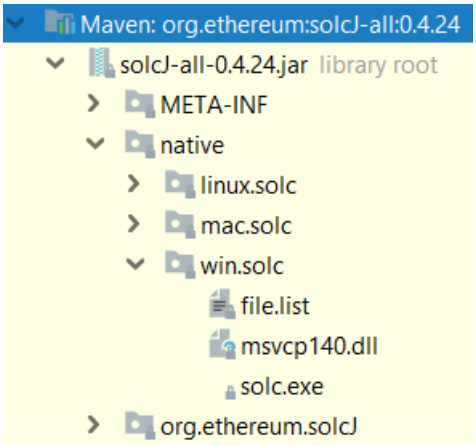
- Identified design patterns in Smart Contracts[1]
  - Check the pattern "**ORACLE (DATA PROVIDER) PATTERN**":
    Problem: An application scenario requires knowledge contained outside the blockchain, but Ethereum contracts cannot directly acquire information from the outside world. On the contrary, they rely on the outside world pushing information into the network.
    Solution**:** Request external data through an oracle service that is connected to the outside world and acts as a data carrier.
  - The Oracle pattern will be replaced later on during the course with a Smart Oracle approach, for example:
    - Centralized Smart Oracle: Provable (ex Oraclize), http://www.oraclize.it/
    - Decentralized Smart Oracle: ChainLink, https://chain.link/
- Secure and reusable Smart Contracts: https://openzeppelin.com/ (check Products -> Contracts)
- ConsenSys best practices: https://consensys.github.io/smart-contract-best-practices/
- Comprehensive guide: https://yos.io/2019/11/10/smart-contract-development-best-practices/
- Handy DApp design patterns: https://medium.com/@i6mi6/solidty-smart-contracts-design-patterns-ecfa3b1e9784

ESSENTIALS

- ETH Gas station:
  - Mainnet: https://ethgasstation.info/
  - Rinkeby: https://www.rinkeby.io/#stats
  - Ropsten: https://ropsten-stats.parity.io/
  - Kovan: N/A from the official Web page: https://kovan-testnet.github.io/website/
- Ethereum explorers:
  - Mainnet: https://ethplorer.io/ https://etherscan.io/
  - Rinkeby: https://rinkeby.etherscan.io/
  - Ropsten: https://ropsten.etherscan.io/
  - Kovan: https://kovan.etherscan.io/
- Web tools:
  - Remix IDE: https://remix.ethereum.org/
  - Oyente: https://oyente.melonport.com/
  - Truffle Suit: https://www.trufflesuite.com/
  - Metamask: https://metamask.io/
- How to compile *solidity* Smart Contracts on preferred OS?
  - Use Maven dependency https://mvnrepository.com/artifact/org.ethereum/solcJ-all
    1. Setup a Maven Project in a local IDE (e.g. Intellij Idea).
    2. Use the solcJ-all dependency for a preferred version.

[1] M. Wöhrer and U. Zdun, "Design Patterns for Smart Contracts in the Ethereum Ecosystem," 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), Halifax, NS, Canada, 2018, pp. 1513-1520.
doi: 10.1109/Cybermatics_2018.2018.00255, URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8726782&isnumber=8726472

3. In the external libraries you can find fully working compilers for Linux, Max and Windows as presented on the figure bellow.



4. You are able to compile it locally, for example in MS Windows:
   solc.exe --bin <mySmartContract.sol> --abi --optimize -o <path for the BIN and ABI output> --overwrite

- Ways to connect to the Ethereum node?
  - Run your own node, guides:
    1. https://dev.to/eric_khun/a-beginner-guide-to-setup-an-ethereum-full-node-4d9
    2. https://medium.com/quiknode/run-your-own-ethereum-node-5c3061925e6a
    3. By running a full node you are able to use filters and events: https://web3j.readthedocs.io/en/latest/filters_and_events.html
  - Use a faucet such as Infura has limitation with events (link). Notify me if this limitation is solved.
  - Run the node in Docker Container, for example: https://hub.docker.com/r/ethereumex/geth-node

PROOF OF CONCEPT PAYMENT EXAMPLES

Bellow there are two types of monetization involving two stakeholders: *end-user* and *service/system entity*.

Fixed price is a basic monetisation use case. The system offers the usage of the service on demand for a price and period, both fixed. After the system deploys the Smart Contract instance, the primary end user requests the session by triggering the respective function and thus sending ETH funds. Upon successful payment, the event notifies the system and initialises the Docker container instances for task X.

Dynamic price complements the fixed price use case and involves the same stakeholders (service entity and primary end user). Its dynamic pricing offers higher flexibility based on the actual service availability. For example, it can be used when the end user knows the maximum time needed for a session of a service. After an agreement is reached, the user pays the full price. The ETH funds are locked by the Smart Contract. The funds may be unlocked to allow for extra payment, if the maximum period is reached and the video conference is continued, or if the end user stops using the service instance by triggering the Smart Contract stop function. In the unlocking phase, the actual usage time is charged - the proportional ETH funds of unused time is returned to the end user, while the rest is sent to the service owner.

Both proof of concept Smart Contracts can be found on the Učilnica Web page:

- FixedPrice.sol
- DynamicPrice.sol

Feel free to adopt, fix and enhance the presented Smart Contracts.

Author: Assist. Sandi Gec, sandi.gec@fri.uni-lj.si