# Practical Project: Build a DApp in Solidity + Web3

Implement a **fully-functional DApp (decentralized application)** consisting of **Solidity smart contract** in the Ethereum blockchain + **client-side app (UI)** and optionally a server-side logic. The app should keep some **data** + **logic** in the Ethereum blockchain and should provide **UI** to access it.

## Project Description

You are **free to decide** what DApp to implement. It might be for example a decentralized e-commerce app, decentralized payment system, decentralized trading app or any other decentralized application or system.

Design and implement a simple **fully-functional DApp** (decentralized application) consisting of:

* **Smart contract**(s), implemented in Solidity, deployed in the **Ropsten testnet**
* **Client-side app** (Web, desktop or mobile), optionally with built-in **wallet**
* Optionally, use a **server-side logic** (Web app / Web services / other components)
* Optionally, use **decentralized storage** like IPFS, Swarm, Storj, Sia, MaidSafe or other

## Project Ideas

You are free to develop **your own idea**, but you might be **inspired by the ideas below**:

### Decentralized Marketplace

* **Sellers** register and list items for sale in the **Ethereum blockchain**.
  + Each item consists of **JSON document** + **images**, stored in **IPFS**.
  + Each seller holds in a **smart contract** a list of **IPFS hashes** of the items for sale.
* **Visitors** can **view** the sellers and their items listed for selling.
  + Visitors don’t see the sellers’ contacts and address.
* Visitors can **register** as **buyers** and **purchase** items.
  + When an item is purchased, the **payment is sent to the smart contract**.
  + The smart contract sends 95% to the seller and 5% retains as commission.
* **Sellers** should have **Ethereum address**, do not need to have a wallet.
* **Buyers** should have a **wallet**. It might be a client-side JS wallet or MetaMask. Transaction signing should happen at the client-side.

### Ponzi Contract

* The **Ponzi contract** is a **smart contract** where investors put funds and get **interest** from their funds staying in the Ponzi contract. At payout, a waiting period is enforced, or penalty applies.
* Each investor can have **multiple investments**. Each investment is managed separately.
* For each investment, the investor takes an **interest, increasing over the time**.
  + After the 1st month 🡪 1%
  + After the 2nd month 🡪 2%
  + After the 3rd month 🡪 4%
  + After the 4th month 🡪 4%
  + After the 5th month and after each next month 🡪 5%
* For each investment, its funds (the investment + the interest gained) are **locked in the smart contract**.
* If an investor wants a **payout**, the rules are as follows:
  + Immediate payout 🡪 get 50% of the funds
  + Payout after 1 month 🡪 get 60% of the funds
  + Payout after 2 months 🡪 get 70% of the funds
  + Payout after 3 months 🡪 get 80% of the funds
  + Payout after 4 months 🡪 get 90% of the funds
  + Payout after 5 months 🡪 get 80% of the funds
  + At any time, an **investor can request payout**, which will execute as described above. An internal counter will execute the payout at the same day a few months later.
  + If a new payout request is entered, and existing payout request is already active, **the new request replaces the old**, and the time counting is reset.
  + After the delay period is passed, the investor can **execute the payout**.
* At any time, if the smart contract runs **out of funds**, it may **declare bankruptcy** and stop paying payouts.
* As **new investments are coming**, the contract will be able to **execute payouts**. If the new investments stop or slow down, the contract will be unable to execute payouts at some time. It is a Ponzi scheme, but no hidden (as it usually is implemented), but public, declared and accepted by all investors.
* Implement a **client-side app (UI)** to **browse the investments** and display information and predictions by choice, based on the financial model, described above.
* Implement **UI** for **entering new investments** and for **requesting a payout** and **executing a payout** (after the waiting time is passed). You may use **MetaMask** or **client-side wallet**.

### Buy Crypto All-Stars Cards

* **Buy a Star** is a **Crypto All-Stars Cards** platform whichallowsusers to buy and sell the only **Smart Contract** in existence of their favorite Crypto Star personalitiesusing the blockchain.
  + Sellers register and list items for sale in the Ethereum blockchain
* Each **Crypto All-Star Card** is linked to only one **Smart Contract Token** on the Ethereum blockchain.
  + Use a browser like Chrome, Firefox or Brave
  + Get the metamask browser extension <https://metamask.io/>
  + Visit a website, where the buying and selling will be executed, it could be anything like buyacryptostar.io
  + Select and purchase your favorite **Crypto All-Star** character
* To purchase a **Smart Contract** send Ether to the contract using Metamask. If someone wants to buy one of your current contracts, the buyer has to pay you more than the amount of your original purchase.
* When a **Crypto All-Star Card** (Smart Contract) is purchased, you own the only one in existence. The value of the **Crypto All-Star Card** is automatically re-listed at a higher price. Therefore anyone can buy your card by paying a new and higher list price.
* The price of each contract increases with each transaction.
* If you buy a contract for 0.03 ETH, another player can take it away from you. If that happens then you will automatically receive 0.06 ETH! Contracts double in price with each transaction until they reach 0.07 ETH.
  + Price increase example: 2x from 0 ETH to 0.03 ETH, 1.2x from 0.05 ETH to 0.5 ETH and 1.15x from 0.5 ETH up.

## Technologies

Technologies for the **blockchain** components of the DApp:

* Use **Ethereum** and **Solidity** for the smart contracts. Deploy your contracts in the **Ropsten testnet**.
* Optionally, use **Truffle** to organize your development and deployment process for the smart contracts.
* Connect to the blockchain through **local Ethereum node** (e.g. geth or Parity) or through a public JSON RPC endpoint like **Infura** (<https://ropsten.infura.io>) or through **MetaMask**.
* Use **Web3.js** or any other library / component (like Nethereum, Web3j or web3.py) to connect to the Ethereum network.
* For security reasons, it is recommended that all **transactions** to the smart contracts to be **signed inside the client-side app** and the user’s private keys to never leave their app.
* You may use a **wallet library** / API / component, e.g. BIP32 / BIP39 / BIP44 HD wallet library.
* Optionally, use **cryptographic libraries** of choice.

For the **client-side app** and for the **server-side logic**, you may use **technologies and platforms of your choice**, like JavaScript and HTML5, Node.js, Java, Android, iOS, Swift, C# and .NET Core, Python, or any other.

## Work Individually

Work **individually**, not in a team. This project will assess your personal skills. Use **Git** and **GitHub**.

* Use **Git** as source control system for your project development.
* Use **GitHub** as your source-code repository and project management tool.
* Your **source code** should be published as **open-source project** in Internet.

## Forbidden Techniques and Tools

* Your project should be created mainly by **you**.
  + You are **not allowed to copy a project from Internet** and present it as your development.
  + If you **fork code** from from Internet, please **declare this explicitly**.
  + Your contribution should be **significant**, not just clone + adjust or copy / paste.
* You can use external libraries, frameworks, platforms and tools, but **not to clone an entire project** and present it as yours.

## Commit Logs

* You should have **at least 5 commits** (changesets) in the project repository.
* Please **commit every day** during the project development to demonstrate your work progress.
* More commits (especially in more than the last 1-2 days) are valued higher during the project assessment.

## Deliverables

Submit the **URL of your project source code** as deliverable, e.g.

* <https://github.com/SoftUni/SoftUniChain>

Put the following assets in the project repository:

* The complete **source code** of your project (Solidity code, JS / C# / Java / Python code, HTML, CSS, images, scripts, build files, config settings and other files).
* A link to the **smart contracts** in the Ethereum testnet.
* Any other project assets (optionally): documentation, design documents, tests, etc.

## Public Project Defense

Each participant will have to deliver a **public defense** of its work in front of the course trainers and colleagues.

Each participant will have **15-20 minutes** for the following:

* **Demonstrate** the application’s functionality (very shortly).
* Explain the project **architecture**.
* Explain the **smart contracts** and how they work.
* Show the **source code** and explain briefly how it works.

Hints for better presentation:

* Be **well prepared** for presenting maximum of your work for minimum time.
* **Open and start all project assets** beforehand to **save time**, e.g. deploy your smart contracts, create accounts with enough ETH to pay the transaction gas, start your apps, run your server infrastructure, etc.
* Test how to connect your laptop with the **multimedia projector** before the defense to save time.

## Assessment Criteria

Each participant will get **individual score** during the evaluation, depending on the project quality, presentation and effort put in the project. The **score** (0…30) will be sum of the following components:

* **Smart contracts**: 0…**10** score
* **Apps interacting with the contracts**: 0…**10** score
* **Other components**: 0…**5** score
* **Bonus**: 0…**5** score