→ steps ~~fo~~ for smart contracts

! Copy & paste Tailwind config

! Copy package.json of smart-contracts

☆ To create a basic structure of our smart contract we are going to use hardhat
hardhat is ethereum development environment & it allows you to run solidity locally. So, it allow us to test our smart contract first before deploying it.

☆ npx hardhat
choose first option
Press enter & again press enter

☆ Now "npx hardhat test" to check everything is working fine

☆ Make sure to install solidity extension

☆ delete ./contracts/greeater.sol

☆ Create Transactions.sol in contracts
⇒ first "pragma solidity ^0.8.0;" → to choose solidity version we want to work on

contract Transactions{ // same as classes in oop
→ contract name
}

A contract in the sense of solidity is a collection of code (its functions) and data (its state) that resides at a specific address on the Etherium blockchain

UINT256 VariableName → Creating a variable
Unsigned integer 256 bits in size

parameters
event Transfer(address from); // function
type variable name

struct TransferStruct{ // object
address sender;
unit amount;
}
what properties it has & there type

TransferStruct[] transactions; → array of type TransferStruct

function addToBlockchain() public{
}
→ functionname
→ visibility of function same as OOPS

```
function getAllTransactions() public view returns ~~(Transferst~~
  (TransferStruct[] memory)~~&~~ {

}
```

→ means this function is going to return array of TransferStruct from memory

memory is a keyword used to store data for the execution of a contract. It holds functions argument data & is wiped after execution.

Storage → default solidity data storage. It holds data persistently & consumes more gas.

payable → when writting a smart contract, you need to ensure that money is being sent to the contract and out of the contract as well. Payable does this for you, any function in solidity with the modifier payable ensures that the function can send & receive ether. Additionally, if you want a function to process transactions and have not included the payable keyword the transaction will be rejected automatically.

→ Goto scripts folder
rename file to deploy.js
make an arrow function

⇒ install metamask extension in your browser

- → click on show/hide
- → turn on show test network
- → click on Etherium Mainnet choose Ropsten Test Network
- → copy your account address
- → goto Ropsten testnet Faucet & paste your address & click on give me & paste for atleast 30 minutes

⇒ goto alchemy & create your account & create a new app & get that app's http address

⇒ goto hardhat.config.js
delete everything
add require("@nomiclabs/hardhat-waffle");

⇒ waffle → plugin to build smart contract tests

⇒ open your hardhat.config.js & replace it with yours. Make sure you don't use mine url & account
Replace url with your http address of your ~~add~~ app given by alchemy
Replace accounts with your account's private key
to get that go to account details in your wallet

⇒ Run npx hardhat run scripts/deploy.js --network ropsten

=) copy transaction address

=) go to your client side make new ~~fla~~ folder utils in ~~the~~ src
make constants. js
Export const ~~cont~~ contractAddress = 'paste your address';

=) in utils folder also create Transactions. json & in this we gonna paste our transactions. json from contracts/transactions.sol

↓

This holds an abi & abi is contract application binary interface that is the standard way to interact with contracts in etherium ecosystem both outside ~~the~~ of blockchain & for contract to contract interactions. This contains all the information about our specific smart contract

=) back in constants import abi

=) created new folder ~~the~~ context in src in client
we gonna use react context api to connect with blockchain

★ Providers -> The ethereum ecosystem provides many methods of interacting with the blockchain. In ethers.js we expose a Provider API that covers the breadth of operations

Date ....

* Metamask → The metamask plug-in enables ethereum for the chrome browser, making it easy for people new ecosystem to get started, exposing the ethereum network as standard web3provider.

```
// metamask injects a web3provider as "web3.current
// Provider", so we can wrap it up in the ethers.js
// web3Provider, which wraps a web3provider and
// exposes the ethers.js Provider API.

const provider = new ethers.providers.Web3Provider
                      (web3.current Provider);

// There is only ever up to one account in Metamask
   exposed
   const signer = provider.getSigner();
```

Connecting to metamask

⇒ metamask injects a global api into websites visited by its users. This API allows website to request users Ethereum accounts, read data from blockchains the user is connected to and suggest that user sign messages & transactions.