A decorative graphic on the left side of the slide, consisting of a network of white lines and circles on a blue gradient background, resembling a circuit board or a neural network.

BLOCKCHAIN: SMART CONTRACTS

LECTURE 9 - DECENTRALIZED APPLICATIONS

FLORIN CRACIUN

IMPORTANT

**Some of the following slides are the
property of**

**Dr. Emanuel Onica & Dr. Andrei Arusoaie
Faculty of Computer Science,**

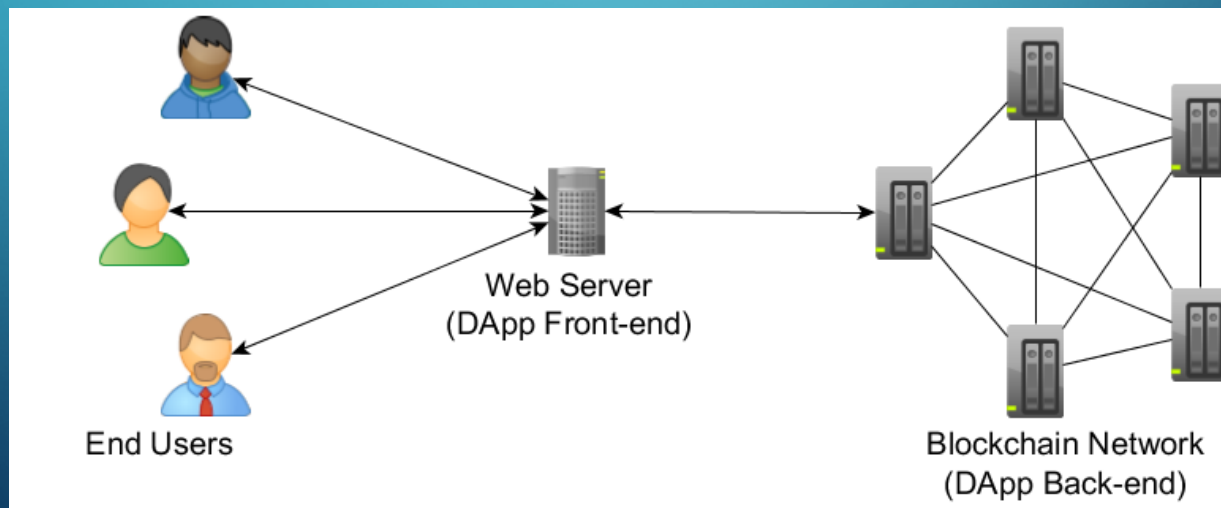
**Alexandru Ioan Cuza University of Iași
and are used with their consent.**

CONTENTS

- 1. What is a DApp?**
- 2. Developing Ethereum DApps:**
 - Tools of the trade
 - Truffle
 - Let's recap some JS asynchrony
 - A bit about Web3
 - MetaMask interaction
- 3. A simple DApp example**
- 4. A very brief overview of meta-transactions**
- 5. Some Dapp trivia**

WHAT IS A DAPP?

- DApp - Decentralized Application:
 - Web application
 - Back-end: smart contracts deployed on a blockchain network (no central authority by nature)
 - Front-end: web-based interface to contracts



WHAT IS A DAPP?

	Web Application	Ethereum DApp
Logic	Web app code	+ Contract code, DApp JavaScript
Data storage	Database	+ Blockchain storage, Swarm
Presentation	HTML, CSS, JavaScript	
Interaction	HTTP, HTTPS	+ Whisper
Deployment	Web server	+ Ethereum network node

DEVELOPING ETHEREUM DAPPS

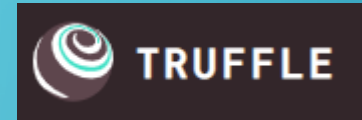
A list of main steps:

1. Implement the smart contracts in Solidity
2. Compile the smart contracts using solc (can be done via Remix, VSCode, etc.)
3. Design the presentation front-end using HTML, CSS, JS
4. Implement the needed DApp JS scripts for interacting with the contracts via the Web3 JS API
5. Deploy the contracts on the Ethereum blockchain
6. Deploy the DApp JS and presentation front-end on a web server (can reside on a blockchain node host too)
7. Serve the DApp

You know steps 1 & 2 already.

ETHEREUM DAPPS: TOOLS OF THE TRADE

Truffle:



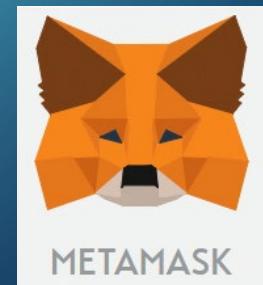
- Automates the DApp development flow
- Integrated smart contract compilation, linking and deployment
- Built-in test environment with interactive console for direct contract communication via Web3 JS API

Ganache:



- Blockchain network simulator with preset accounts and instant transactions

MetaMask:



- Browser plugin available for Chrome, Brave, Firefox
- Permits importing user Ethereum accounts and conducting transactions
- Relies on a provider API which permits websites (i.e., DApps) to interact with the blockchain environment

ETHEREUM DAPPS: TRUFFLE

- Truffle provides a series of commands for automating development operations (there's no actual IDE)
- Truffle installation: `npm install -g truffle`
- Creation of a new project: `truffle init`
- Resulting project structure:
 - `./contracts` – Folder for Solidity contracts
 - `./migrations` – Folder for deployment scripts
 - `./test` – Folder for test files (both `.sol` and `.js`)
 - `truffle-config.js` – Configuration file

ETHEREUM DAPPS: TRUFFLE

truffle-config.js:

- Various configuration options, most of them explained in the commented part
- `networks`: probably the main one typically needing attention – configures settings for network deployment
- Typical setting for local deployment (using Ganache):

```
module.exports = {  
  networks: {  
    development: {  
      host: "127.0.0.1",  
      port: 7545,  
      network_id: "*" // Match any network id  
    }  
  }  
};
```

ETHEREUM DAPPS: TRUFFLE

- Compiling contracts: `truffle compile`
- Compiles all contracts located in `./contracts` (compiler version can be set in `truffle-config.js`)
- One pre-defined contract — `Migrations.sol` — required for keeping track of deployment of the other contracts
- Resulting artifacts (json files including bytecode and contract ABI) will be placed in: `./build/contracts`
- These are needed for creating contract instances in the DApp JavaScript

ETHEREUM DAPPS: TRUFFLE

- **Deploying contracts:** `truffle migrate`
- **Option to specify the network to be chosen from the config:** `--network [network-name]` (default is typically „development“)
- **Executes deployment scripts located in `./migrations`**
- **Scripts are executed in their lexical name order**
- **Each new migrate comand call will resume deployment after previous last executed script**
- **There is one pre-defined initial migration script: `1_initial_migration.js` for deploying the pre-defined `Migrations.sol` contract**

ETHEREUM DAPPS: TRUFFLE

- Adding a typical deployment script (e.g., `2_deploy_contracts.js`):

```
var MyContract = artifacts.require("MyContract");

module.exports = function(deployer) {
    deployer.deploy(MyContract);
};
```

- `artifacts.require()` used to include the contracts for deployment, specified by their name
- The script must export a function with a `deployer` parameter that includes the deployment code
- `deployer` is the object provided by Truffle for deployment related operations

ETHEREUM DAPPS: TRUFFLE

Functions provided by deployer:

- `deploy(contract, args..., options)` – deploys the contract initializing the constructor with any provided args; `options` is optional and can include:
 - `gas` – amount spent for contract deployment
 - `from` – origin address that pays for deployment (default: first address in `eth.accounts` objects)
 - `overwrite` – boolean specifying if new deployment should „replace” a previous one; if true, a new contract version will be deployed and address will be updated in corresponding `./build/contracts` artifact
- `link(library, dependent_contracts)` – links a deployed library contract to one or more contracts that will use the library
- `then(function())` – allows executing a custom function once a contract is deployed for specific setup; works like a JS promise

ETHEREUM DAPPS: TRUFFLE

Interacting with contracts:

- `truffle develop` provides a console interaction environment
- This integrates with a Truffle provided blockchain network (similar to Ganache; default port 9545)
- `truffle console` provides a similar interaction environment but with the configured development network (e.g., provided through Ganache)
- Interaction with contracts via the Truffle console is done using a Web3 JS API provided by Truffle
- Note that the Truffle provided API is not completely compatible with the latest official Web3 JS API version! (e.g., access to contract methods is simplified in the Truffle version)

Debugging contracts: `truffle debug`

More info on the Truffle website

ETHEREUM DAPPS: TRUFFLE

- Until now we've seen more or less the same operations that we've also seen available in previous weeks by using Remix

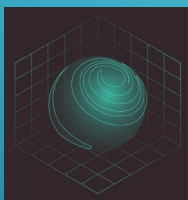
	Truffle	Remix
Integrated contract editor	No	Yes
Contract compilation	Yes	Yes
Contract deployment	Yes (with complex linking)	Yes
Contract debugging	Yes	Yes
Contract interaction	Yes	Yes
Blockchain test network	Yes	Yes
Dev friendly	Console	Web Browser
DApp JS development and integration	via Truffle Boxes (since Truffle 3.0)	via Pipeline (WiP Remix plugin)

- Truffle *currently* provides a more mature option, but keep in mind that alternatives are there

ETHEREUM DAPPS: TRUFFLE

Integrating the DApp front-end and JavaScript for contract interaction:

- Directly using Truffle until version 3.0
- Since version 3.0 separated via...



TRUFFLE BOXES

- Boilerplate packages added to basic Truffle project
- A variety of boxes officially and custom designed providing diverse DApp resources
- Creating a project from a box: `truffle unbox box-name`

ETHEREUM DAPPS: TRUFFLE

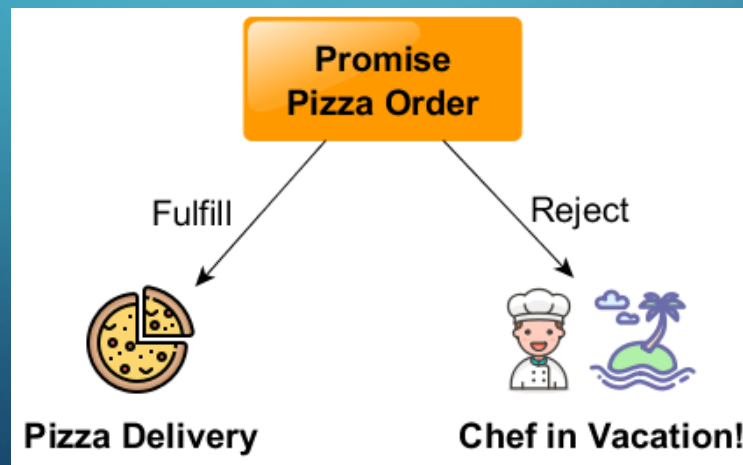
- The *webpack* box: probably the simplest option for adding needed DApp JS to what we have until now
- `./app` – extra directory created after unboxing
- `./app/src` – JS source code and HTML presentation part should be placed here
- `npm run dev` (run in app folder):
 - „builds” the JavaScript code
 - starts a development web server
 - serves the DApp (default - localhost:8080)

ETHEREUM DAPPS: RECAP SOME JS ASYNCHRONY

- Because the current main way to interact with a contract via a DApp is through the Web3 API:
 - which is written in JavaScript
 - which uses plenty of asynchronous calls
 - which can be tricky
- No previous experience in JavaScript?
- **Strong recommendation: quickly find a tutorial**

ETHEREUM DAPPS: RECAP SOME JS ASYNCHRONY

- JavaScript Promises
 - Objects representing the outcome of an operation
 - Can be in one of three states:
 - Pending: not completed
 - Fulfilled: completed and having a resolved value
 - Rejected: failed and having a reason for failure



ETHEREUM DAPPS: RECAP SOME JS ASYNCHRONY

JavaScript Promises

- **Instantiating a promise:**
 - `const myPromise = new Promise(myFunction);`
- ***myFunction* is a function that has two pre-defined function parameters (not provided by the developer):**
 - ***resolve*** – a function receiving one argument that will be set as resolved value for the promise
 - ***reject*** – a function receiving one error argument that will be set as failure reason for the promise
 - **Example:**

```
function myFunction(resolve, reject) {  
    if (chef === available) {  
        resolve('Bon Appetit!');  
    } else {  
        reject('Starving!');  
    }  
}
```


ETHEREUM DAPPS: RECAP SOME JS ASYNCHRONY

- JavaScript Promises

- Consuming a promise:

- When a promise is settled – either fulfilled or rejected, its result is ready to be „consumed”
 - `.then(successFunction [, failFunction])` called on the promise; executes the passed callback functions, which receive as argument the promise result in case of success or, respectively on failure
 - `.catch(failFunction)` called on the promise executes a callback function, which receives as argument the promise rejection reason
 - `.then` and `.catch` **are asynchronous** and also return promises having the settled result value of the initial promise

ETHEREUM DAPPS: RECAP SOME JS ASYNCHRONY

- **JavaScript Promises**

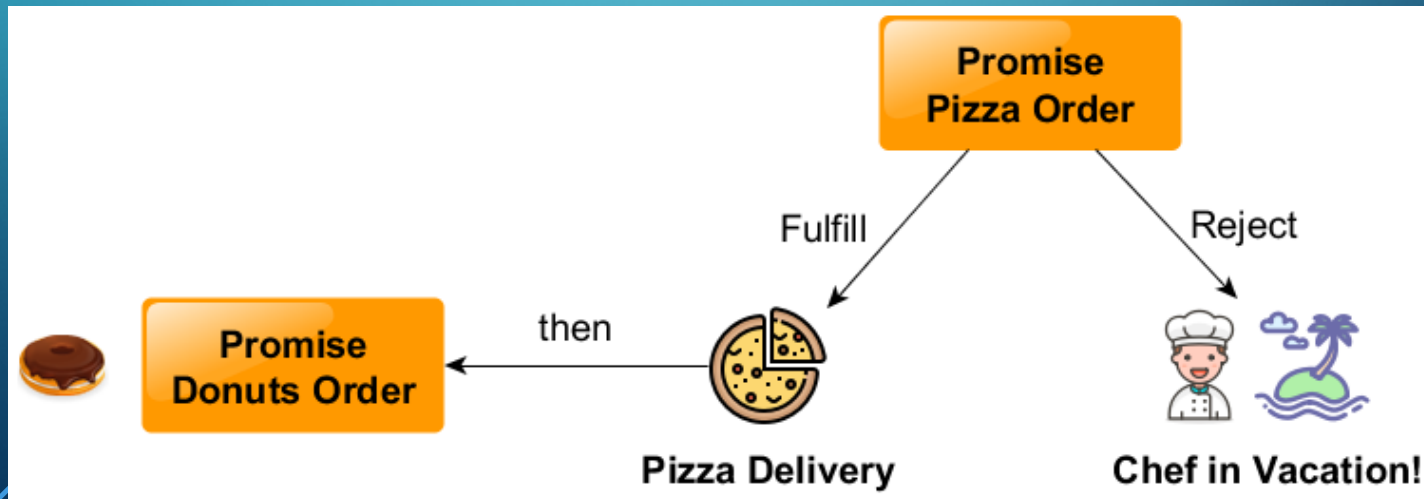
- **Example:**

```
function successFunction(fulfilled) {  
    console.log('Success ' + fulfilled);  
}  
  
function failFunction(rejected) {  
    console.log('Fail ' + rejected);  
}  
  
const myPromise = new Promise(myFunction)  
    .then(successFunction)  
    .catch(failFunction);
```

ETHEREUM DAPPS: RECAP SOME JS ASYNCHRONY

• Chaining Promises

- Promises can be chained: a new Promise can be created as „response” for settling a current one
- Chaining is done by instantiating new Promises in the *then* callback functions
- Returning the new Promise replaces the first Promise settled result with the new one



ETHEREUM DAPPS: RECAP SOME JS ASYNCHRONY

Chaining Promises – Example:

```
function donutsFunction(resolve, reject) {  
    if (donutsChef === available) {  
        resolve('Full Meal Done!');  
    }  
    else { reject('Starving!'); }  
}  
  
function successFunction(fulfilled) {  
    console.log('Success '+fulfilled);  
    return new Promise(donutsFunction);  
}  
  
function fullSuccessFunction(fulfilled) {  
    console.log('Finally...'+fulfilled);  
}  
  
const myPromise = new Promise(myFunction)  
    .then(successFunction)  
    .then(fullsuccessFunction)  
    .catch(failFunction);
```


ETHEREUM DAPPS: RECAP SOME JS ASYNCHRONY

The `async/await` syntax:

- Syntactic sugar built on promises
- Functions can be declared as *async* indicating that these contain asynchronous Promise resolving calls
- *await* can be used to wait (block) until an asynchronous call is finalized
- can be used to simplify the reading of long *then* chains by splitting Promise calls
- Example:

```
async function myFunction() {  
  console.log('Not moving until getting pizza!');  
  let mainCourse = await functionPromiseReturningPizza();  
  console.log('Not moving until getting donuts!');  
  let dessert = await functionPromiseReturningDonuts();  
}
```

ETHEREUM DAPPS: A BIT ABOUT WEB3

- Communication with Ethereum nodes is done through convoluted JSON RPC calls
- *Providers* consist in implementations offered for transport services through the RPC mechanism
- We're not getting into the shady providers world – we'll just use some.

ETHEREUM DAPPS: A BIT ABOUT WEB3

Web3:

- Web3.js is a collection of JavaScript libraries that provides a high level API for interacting with an Ethereum node via a Web3 provider
<https://web3js.readthedocs.io/en/v1.3.0/>
- Web3.js API is agnostic of the Web3 provider's transport implementation (HTTP, web sockets, etc.)
- By far the main solution currently used for this purpose (alternative ethers.js much less used)
- Various versions or adaptations embedded in other software (e.g., Truffle, MetaMask, etc)
- Quite volatile API – currently moving towards v2.0 announced to include breaking changes

ETHEREUM DAPPS: A BIT ABOUT WEB3

- Split in four main modules:
 - **web3.eth** – main interaction with blockchain and contracts
 - **web3.utils** – helper functions
 - **web3.shh** – whisper communication protocol API
 - **web3.bzz** – swarm storage protocol API

- Instantiation of a new Web3 object:

```
//server side (using node.js module)
```

```
var web3 = require('web3');
```

```
//client side (injecting web3 in html source)
```

```
<script src=" [url to]/web3.js"></script>
```

```
var web3 = new web3(provider);
```

```
// e.g., for MetaMask provider: window.ethereum
```

```
// for Ganache provider: "ws://localhost:7545"
```


ETHEREUM DAPPS: A BIT ABOUT WEB3

- Getting the account addressess controled by the node:
 - `web3.eth.getAccounts([callback])`
 - returns a Promise resolving to an Array of addresses
 - optional callback function receives 2 parameters:
 - 1st set to an error if the Promise is rejected
 - 2nd set to the function result if the Promise fullfills
 - such callback provision as usual pattern for most methods in Web3.js

- Example:

```
let accounts = await web3.eth.getAccounts((err, accs) =>{ if (err) {  
  console.log('Error'); }  
    else { console.log('No.of accounts'+accs.length); }  
  });  
// await call should be done in an async function
```

ETHEREUM DAPPS: A BIT ABOUT WEB3

- Getting the balance of an account:
 - `web3.eth.getBalance(address [,block][,callback])`
 - returns a Promise resolving to the address balance
 - block can specify a different chain block than the last default one
 - optional callback function as before
- Instantiating a contract variable for interaction:
 - `let contract = new web3.eth.Contract(abi, addr);`
 - *abi* is the JSON artifact generated after contract compilation
 - *addr* is the address of the deployed contract
 - additional options object parameter can specify {from:, gasPrice:, gas:, data:} (data used just when deploying)

ETHEREUM DAPPS: A BIT ABOUT WEB3

- Calling a „constant” (view) contract method:
 - `contractInstance.methods.theMethod([params]).call(options [,callback])`
 - returns a Promise resolving to the method's return value
 - *params* specify the method's parameters
 - *options* is an object that can specify {*from*:, *gasPrice*:, *gas*:}
 - optional callback function fired with error and method result parameters
- A *call* does not perform an effective transaction on the contract
- A *call* cannot alter smart contract state
- Can be used also for checking the value of public data members of the contract (similar to getters) using the name of the member as *theMethod*

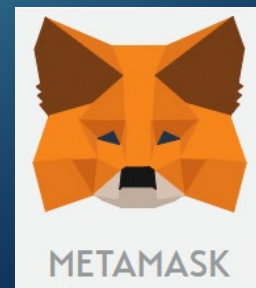
ETHEREUM DAPPS: A BIT ABOUT WEB3

- Executing a contract method as transaction:
 - `contractInstance.methods.theMethod([params]).send(options [,callback])`
- Callback function is fired with error and transaction hash parameters
- Returns a PromiEvent – promise with „event emitter”, resolving eventually to a *receipt* object with event names as keys and events data as properties
- *params*: options as in the *call* case
- additional fired events can be caught with `.on(event)`:
 - `ctrInstance.methods.theMethod().send(options).on('transactionHash', function(hash) {
/* fired when the transaction hash is available */
on('confirmation', function(confNr, receipt) {
/* fired at each block confirmation up to 24 */
on('receipt', function(receipt) {
/* fired when the transaction receipt is available */`

ETHEREUM DAPPS: METAMASK INTERACTION

MetaMask:

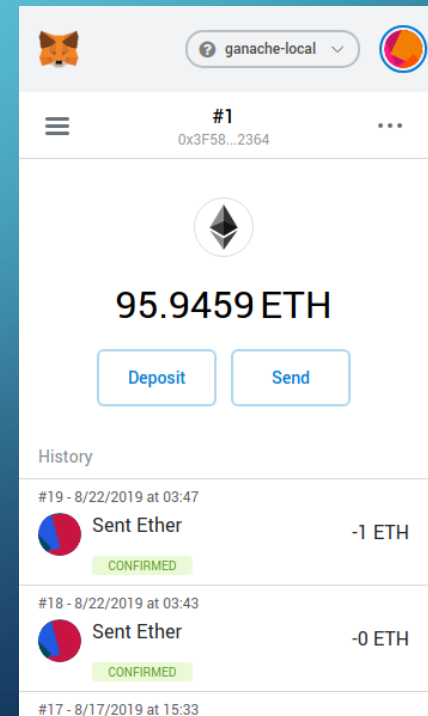
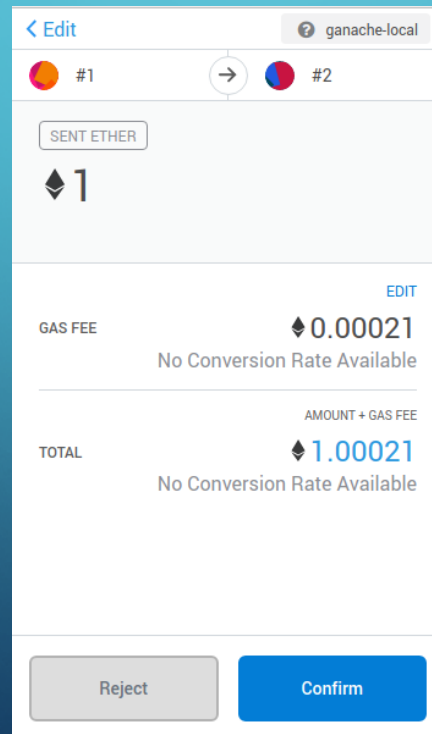
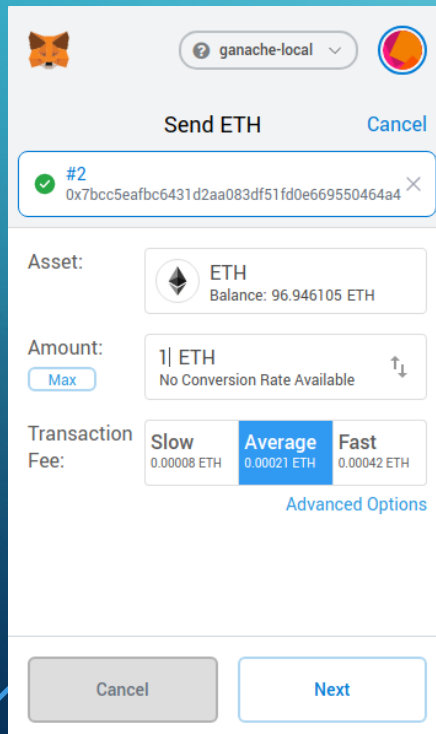
- Browser plugin offering a provider for interacting with the Ethereum network
- Eliminates the need to run a full node with the whole blockchain on the client machine
- Can connect to any Ethereum network (i.e., main, Ropsten, Rinkeby) or a local development blockchain (i.e., Ganache, Truffle develop)
- Works best with Chrome and Brave browsers



ETHEREUM DAPPS: METAMASK INTERACTION

MetaMask:

- Can also be used as a standalone wallet
- Permits importing user accounts and performing simple Ether transactions to other user or contract accounts



ETHEREUM DAPPS: METAMASK INTERACTION

Interacting with MetaMask in the DApp JS code:

- MetaMask injects its provider object: *window.ethereum* in the pages visited in browser

- Simple init of the provider in the code using Web3:

```
const ethEnabled = () => {  
  if (window.ethereum) {  
    window.web3 = new web3(window.ethereum);  
    return true;  
  }  
  return false;  
}  
  
if (!ethEnabled()) {  
  alert("Browser not Ethereum-compatible!");  
}
```

- Metamask suggested init (multiple changes during latest year) and more info on specific provider usage: <https://docs.metamask.io/guide/ethereum-provider.html>

A SIMPLE DAPP EXAMPLE

- A simple DApp example: the *CarAuction* application

- General info defined in a base contract, e.g.:

```
enum auction_state{ CANCELLED,STARTED}

struct car{ string  Brand; string  Rnumber;}

car public Mycar;

address[] bidders;

mapping(address => uint) public bids;

auction_state public STATE;

uint256 public auction_start;

uint256 public auction_end;

uint256 public highestBid;

address public highestBidder;

modifier an_ongoing_auction() {

    require(block.timestamp <= auction_end &&

        STATE == auction_state.STARTED);

    _;

}
```


A SIMPLE DAPP EXAMPLE

- Particular derived contract implementing specific auction bidding rules
- Constructor in this contract:

```
constructor (uint _biddingTime, address payable _owner,  
            string memory _brand, string memory _Rnumber) {  
    auction_owner = _owner;  
    auction_start = block.timestamp;  
    auction_end = auction_start + _biddingTime*1 hours;  
    STATE = auction_state.STARTED;  
    Mycar.Brand = _brand;  
    Mycar.Rnumber = _Rnumber;  
}
```

A SIMPLE DAPP EXAMPLE

- The principal function in the contract registers a new bid as long as it's higher than all previous bids:

```
function bid()  
public payable an_ongoing_auction override returns (bool) {  
    require(bids[msg.sender] + msg.value > highestBid,  
            "You can't bid, Make a higher Bid");  
  
    highestBidder = msg.sender;  
    highestBid = bids[msg.sender] + msg.value;  
    bidders.push(msg.sender);  
    bids[msg.sender] = highestBid;  
    emit BidEvent(highestBidder, highestBid);  
    return true;  
}
```

- Other methods allow withdrawing bids after auction concludes, canceling or destroying the auction by the owner

A SIMPLE DAPP EXAMPLE

- A very basic HTML presentation layer of the DApp for interacting with the contract (you can beautify this with some CSS):

Car Auction Dapp

Car Details

Brand:Toyota

Registration Number:9753

Bid value

eg. 100

Successfull bid, transaction ID0x894464325a7d8196d6d10619ac32c4e1aa1053a9b2e2413745d69fa6337f0e62

Auction Details

- Auction End: 1604859687
- Auction Highest Bid: 0
- My Bid: 0
- Auction Highest Bider: 0x0000000000000000000000000000000000000000
- Auction Status: 1

Events Logs

Auction Cancelled at 1604856208

Auction Operations

A SIMPLE DAPP EXAMPLE

Some insight in the JavaScript used to interact with the contract:

- Initialization with the MetaMask provider – as presented a couple slides ago
- Using an *init* function in .js to perform various initialization and to populate the web interface, e.g., can bind this to page loading: `window.onload = async function init(){ ...`
- Getting the user accounts (current one will be the bidder):
 - `const accounts = await ethereum.request({ method: 'eth_requestAccounts' });`
 - note that the MetaMask provider will export only one account – the active one (`window.bidder = accounts[0]`)

- Initializing the contract for accessing it:

```
var myauctionContractABI = [the contract ABI];
```

```
var contractAddress = "the contract address";
```

```
window.auction = new web3.eth.Contract(myauctionContractABI,  
contractAddress);
```


A SIMPLE DAPP EXAMPLE

- In .js (also init function) – populating in the web interface elements by retrieving values from the contract storage via the *call* method provided by Web3 API, e.g., :

```
auction.methods.Mycar().call().  
then(function(result){  
    document.getElementById("car_brand").  
        innerHTML=result[0];  
  
    document.getElementById("registration_number").  
        innerHTML=result[1];  
});  
then(function(result){  
    $("#a_address").html(result);  
})
```

A SIMPLE DAPP EXAMPLE

- **Initiating a new bid – in the .js – via the `send` method in the Web3 API:**

```
function bid() {  
    var mybid = document.getElementById('value').value;  
    auction.methods.bid().send(  
        {  
            from: window.bidder,  
            value: web3.utils.toWei(mybid, "ether"),  
            gas: 200000  
        },  
        function(error, result){  
            if(error) {  
                console.log("error is "+ error);  
                document.getElementById("biding_status").innerHTML= "Think to bidding higher";  
            }  
            if (!error)  
                document.getElementById("biding_status").innerHTML=  
                    "Successfull bid, transaction ID"+ result;  
        }  
    );  
}
```

- **Binding the function with the web interface – in the .html:**

```
<button class="btn btn-default" id="transfer" type="button"  
onClick="bid()">Bid!</button>
```

A SIMPLE DAPP EXAMPLE

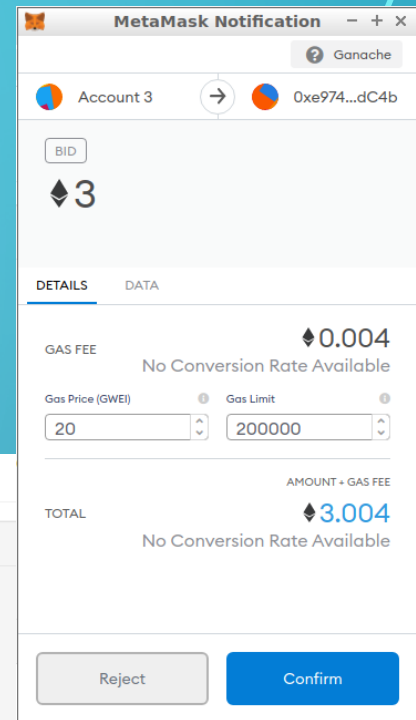
- Catching events thrown by the contract (set in the init function in .js):

```
var BidEvent = auction.events.BidEvent(
    {
        filter: {
            fromBlock: 0,
            toBlock: 'latest',
            address: contractAddress,
            topics: [web3.utils.sha3('BidEvent(address,uint256)')]
        }
    },
    function(error, result){
        if (!error) {
            console.log(result);
            $('#eventslog').html(result.returnValues.highestBidder +
            ' has bidden(' + result.returnValues.highestBid + ' wei)');
        } else { console.log(error); }
    }
);
```

A SIMPLE DAPP EXAMPLE

• After sending a bid:

(note that the web UI is not fully updated until refresh;
you can try to fix this as exercise)



Car Auction Dapp

Car Details

Brand:Dacia

Registration Number:97531

Bid value

eg. 100

3

Bid!

Successfull bid, transaction ID0x7120d0cd5a221f46658b9cb1a1ac576d7a24178e6073a7166ac557e9a926e9a8

Auction Details

- Auction End: 1604848706
- Auction Highest Bid: 0
- My Bid: 0
- Auction Highest Bider: 0x0000000000000000000000000000000000000000
- Auction Status: 1

Events Logs

0xD1Cc93D446A30929f40Ffd9B6f5Ae606563401A2 has bidden(3000000000000000 wei)

Car Auction Dapp

Car Details

Brand:Dacia

Registration Number:97531

Bid value

eg. 100

10

Bid!

Auction Details

- Auction End: 1604848706
- Auction Highest Bid: 3
- My Bid: 3
- Auction Highest Bider: 0xD1Cc93D446A30929f40Ffd9B6f5Ae606563401A2
- Auction Status: 1

Events Logs

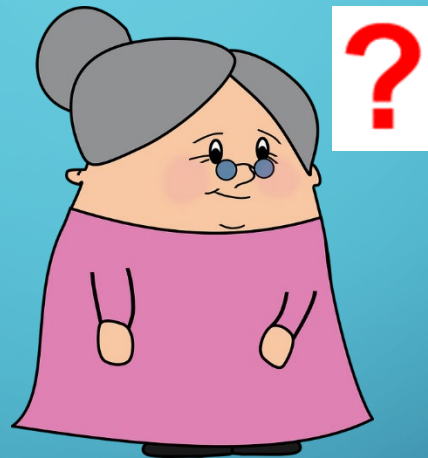
44/49

META-TRANSACTIONS IN THE DAPP CONTEXT

Big idea! - Let's create a wonderful, decentralized, trustworthy DApp for registering energy consumption readings!

The issue of adopting it:

- transactions cost gas
- gas costs Ether
- people don't have Ether



Where to get Ether?

- crypto-exchanges (e.g., Binance: <https://www.binance.com/en/buy-Ethereum>)
- usually requires a KYC process (e.g., Binance: <https://www.binance.com/en/support/faq/360027287111>)

User onboarding in DApps is a cumbersome process

META-TRANSACTIONS IN THE DAPP CONTEXT

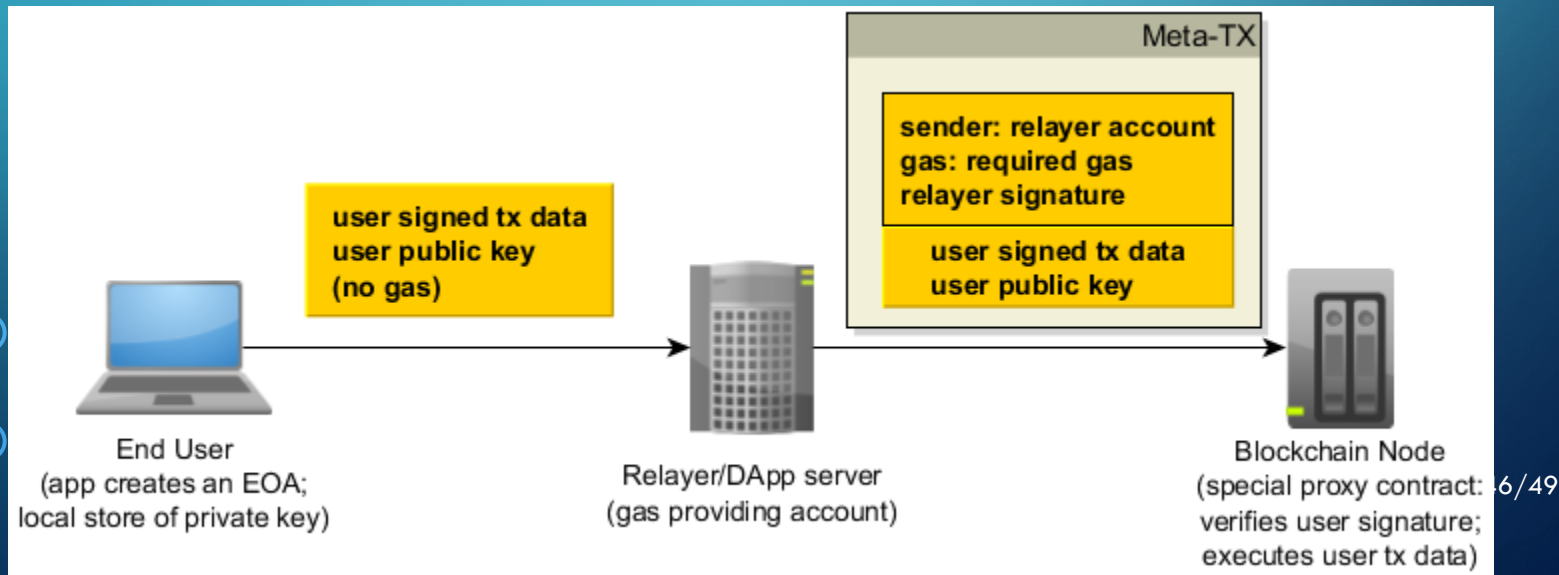
The solution: „meta-transactions”

A generic name for a bunch of approaches:

<https://ethresear.ch/t/native-meta-transaction-proposal-roundup/7525>

Most common idea - using a trusted relayer proxy:

(extremely simplified depiction)



META-TRANSACTIONS IN THE DAPP CONTEXT

Much of the focus in existing implementations set on solving a subproblem:

- permit token owners to do transactions using just their tokens

Some deployed solutions:

- Gas Station Network - <https://opengsn.org/>
- Biconomy - <https://biconomy.io/>

These require specific adaptations of the DApps (i.e., changing backend contract side to support the mechanism).

SOME DAPP TRIVIA

- Various DApp directories online:

<https://dappradar.com/> , <https://www.stateofthedapps.com/> , etc.

- One of most known DApp examples: CryptoKitties



What is CryptoKitties?

CryptoKitties is a game centered around breedable, collectible, and oh-so-adorable creatures we call CryptoKitties! Each cat is one-of-a-kind and 100% owned by you; it cannot be replicated, taken away, or destroyed.

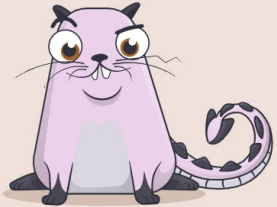
- At some point using more than 6% of the Ethereum blockchain! (cnbc.com December 2017)
- Scope of the game: collect illustrated cats with specific genes, which breed and produce more cats after some periods of time
- DApp offers also trading cats support

SOME DAPP TRIVIA

- Most expensive transaction in September 2018 at 600 ETH (~170.000\$ at the currency rate of the time of sale)


💰 <u>CryptoKitties</u> Sales 💰				
~ Made by @nieldlr 🐱 ~				
Total Sales: 682548			Average Sale Price: \$40.91	
Total Unique Kittens Sold: 524270			Median Sale Price: \$2.71	
Total Ether Sold: 60415.40 ether				
Total USD Sold: \$27922526.61				
Kitty Info		Pricing		Details
ID	Ether	USD (at time of sale)	USD (current value)	Sale Date
<input type="text"/>				
896775	≡ 600.0000	\$172625.79	\$282000	9/4/2018, 7:21:26 AM
18	≡ 253.3368	\$110707.16	\$119068.29	12/7/2017, 5:28:18 AM
4	≡ 247.0000	\$107816.49	\$116090	12/6/2017, 9:41:57 PM



- Cat still/back on sale 😊 in November '20:



Dragon

896775 ⚙ Gen 9 ⌚ Snappy Cooldown (30m)

rabono Owner 

  38 Not in any Collection

Buy now price

600 ETH

\$263k USD

Buy with ETH