Advanced Solidity 1

Cheatsheet : https://docs.soliditylang.org/en/v0.8.11/cheatsheet.html (https://docs.soliditylang.org/en/v0.8.11/cheatsheet.html)
ReadtheDocs documentation (https://docs.soliditylang.org/en/latest/structure-of-a-contract.html) will now open in remix

solidity-upgrade (https://docs.soliditylang.org/en/latest/using-the-compiler.html?highlight=solidity-upgrade#solidity-upgrade)- a tool to automatically apply new language features

Available Upgrade Modules

Module	Version	Description
`constructor`	0.5.0	Constructors must now be defined using the `constructor` keyword.
`visibility`	0.5.0	Explicit function visibility is now mandatory, defaults to `public`.
`abstract`	0.6.0	The keyword `abstract` has to be used if a contract does not implement all its functions.
`virtual`	0.6.0	Functions without implementation outside an interface have to be marked `virtual`.
`override`	0.6.0	When overriding a function or modifier, the new keyword `override` must be used.
`dotsyntax`	0.7.0	The following syntax is deprecated: `f.gas(...)()`, `f.value(...)()` and `(new C).value(...)()`. Replace these calls by `f{gas: ..., value: ...}()` and `(new C){value: ...}()`.
`now`	0.7.0	The `now` keyword is deprecated. Use `block.timestamp` instead.
`constructor-visibility`	0.7.0	Removes visibility of constructors.

For a set of medium articles exploring Solidity see the All about Solidity series (https://jeancvllr.medium.com/all-about-solidity-article-series-f57be7bf6746)

Addresses

Address literals

```
address public constant DAI = 0x6B175474E89094C44Da98b954EedeAC495271d0F;
```

Checksums

Defined in EIP 55 (https://github.com/ethereum/EIPs/blob/master/EIPS/eip-55.md)
Convert the address to hex, but if the ith digit is a letter (ie. it's one of abcdef) print it in uppercase if the 4*ith bit of the hash of the lowercase hexadecimal address is 1 otherwise print it in lowercase.

- Backwards compatible with many hex parsers that accept mixed case, allowing it to be easily introduced over time
- Keeps the length at 40 characters
- On average there will be 15 check bits per address, and the net probability that a randomly generated address if mistyped will accidentally pass a check is 0.0247%.

Address Payable

Since version 0.5.0 there has been a distinction in the code between addresses that can receive ETH and those that can't.
To indicate that ETH can be sent to an address use the `payable` keyword in the variale declaration

```
address payable ethWallet;
```

EOA Address creation

1. Start with the public key (128 characters / 64 bytes)
2. Apply the Keccak-256 hash to the public key
   Keccak256(pubKey) =
   2a5bc342ed616b5ba5732269001d3f1ef827552ae1114027bd3ecf1f086ba0f9
3. Take the last 40 characters / 20 bytes (least significant bytes) of the resulting hash.
   E.g.
   2a5bc342ed616b5ba5732269**001d3f1ef827552ae1114027bd3ecf1f086ba0f9**
   Address = 0x001d3f1ef827552ae1114027bd3ecf1f086ba0f9

Contract Addresses

CREATE gives the address that a contract will be deployed to

```
1 keccak256(rlp.encode(deployingAddress, nonce))[12:]
```

CREATE2 instroduced in Feb 2019

```
1 keccak256(0xff + deployingAddr + salt + keccak256(bytecode))[12:]
```

Function selectors

There is a useful shortcut get the signatures of functions

You can get the function signature of the f() by using
this.f.selector in another function.
selector is a method that returns a value of bytes4.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.14 <0.9.0;

library L {
    function f(uint256) external {}
}

contract C {
    function g() public pure returns (bytes4) {
        return L.f.selector;
    }
}
```

Encoding the function signatures and parameters

Example (https://docs.soliditylang.org/en/v0.6.2/abi-spec.html?highlight=selector#examples)

```
pragma solidity ^0.8.0;

contract MyContract {

  Foo otherContract;


  function callOtherContract() public view returns (bool){
      bool answer = otherContract.baz(69,true);
      return answer;
  }
}


contract Foo {
    function bar(bytes3[2] memory) public pure {}
    function baz(uint32 x, bool y) public pure returns (bool r) {
    r = x > 32 || y;
    }
    function sam(bytes memory, bool, uint[] memory) public pure {}
}
```

The way the call is actually made involves encoding the function selector and parameters

If we wanted to call ***baz*** with the parameters **69** and **true** , we would pass 68 bytes total, which can be broken down into:

1. the Method ID. This is derived as the first 4 bytes of
   the Keccak hash of the ASCII form of the signature baz(uint32,bool).

   0xcdcd77c0:

2. the first parameter, a uint32 value 69 padded to 32 bytes

   0x0000000000000000000000000000000000000000000000000000000000000045:

3. the second parameter - boolean true, padded to 32 bytes

   0x0000000000000000000000000000000000000000000000000000000000000001:

In total

***0xcdcd77c000000000000000000000000000000000000000000
000000000000000000000045000000000000000000000000000000
000000000000000000000000000000000001***

This is what you see in block explorers if you look at the inputs to functions

There are helper methods to put this together for you

```
abi.encodeWithSignature("baz(uint32, boolean)", 69, true);
```

Alternatively you can then call functions in external contracts on a low level way via

```
bytes memory payload =
abi.encodeWithSignature("baz(uint32, boolean)", 69, true);

(bool success, bytes memory returnData) =
address(contractAddress).call(payload);

require(success);
```

For complex details of function argument encoding see documentation (https://docs.soliditylang.org/en/latest/abi-spec.html?highlight=selector#function-selector-and-argument-encoding)

Checking Signatures on chain

See Open Zeppelin (https://docs.openzeppelin.com/contracts/4.x/utilities#cryptography)
ECDSA provides functions for recovering and managing Ethereum account ECDSA signatures. These are often generated via web3.eth.sign, and are a 65 byte array (of type bytes in Solidity) arranged the following way: [[v (1)], [r (32)], [s (32)]].

The data signer can be recovered with ECDSA.recover, and its address compared to verify the signature. Most wallets will hash the data to sign and add the prefix '\x19Ethereum Signed Message:\n', so when attempting to recover the signer of an Ethereum signed message hash, you'll want to use toEthSignedMessageHash.

```
using ECDSA for bytes32;

function _verify(bytes32 data, bytes memory signature, address account)
internal pure returns (bool) {
    return data
        .toEthSignedMessageHash()
        .recover(signature) == account;
}
```

EIP712 and EIP2612

EIP712 a standard for signing transactions and ensuring that they are securely used

Domain separator (for an ERC-20)

```
bytes32 eip712DomainHash = keccak256(
    abi.encode(
        keccak256(
            "EIP712Domain(string name,string version,
            uint256 chainId,address verifyingContract)"
        ),
        keccak256(bytes(name())), // ERC-20 Name
        keccak256(bytes("1")),    // Version
        chainid(),
        address(this)
    )
);
```

Permit Function

The permit function is used for any operation involving ERC-20 tokens to be paid for using the token itself, rather than using ETH.
EIP2612 (https://eips.ethereum.org/EIPS/eip-2612)

EIP2612 adds the following to ERC20

```
function permit(address owner, address spender,
    uint value, uint deadline, uint8 v, bytes32 r, bytes32 s) external
function nonces(address owner) external view returns (uint)
function DOMAIN_SEPARATOR() external view returns (bytes32)
```

Traditional Process = Approve + TransferFrom

The user sends a transaction which will approve tokens to be used via the UI.
The user pays the gas fee for this transaction
The user submits a second transaction and pays gas again.

Permit Process

User signs the signature — via Permit message which will sign the approve function.
User submits signature. This signature does not require any gas — transaction fee.
User submits transaction for which the user pays gas, this transaction sends tokens.

Original introduced by Maker Dao (https://docs.makerdao.com/smart-contract-modules/dai-module/dai-detailed-documentation#3.-key-mechanisms-and-concepts)

In more detail

Taken from permit article (https://soliditydeveloper.com/erc20-permit)

1. Our contract needs a domain hash as above and to keep track of nonces for addresses
2. We need a permit struct

```
bytes32 hashStruct = keccak256(
    abi.encode(
        keccak256("Permit(address owner,address spender,
        uint256 value,uint256 nonce,uint256 deadline)"),
        owner,
        spender,
        amount,
        nonces[owner],
        deadline
    )
);
```

This struct will ensure that the signature can only used for

the permit function
to approve from owner
to approve for spender
to approve the given value
only valid before the given deadline
only valid for the given nonce
The nonce ensures someone can not replay a signature, i.e., use it multiple times on the same contract.

We can then put these together

```
bytes32 hash = keccak256(
    abi.encodePacked(uint16(0x1901), eip712DomainHash, hashStruct)
);
```

On receiving the signature we can verify with

```
address signer = ecrecover(hash, v, r, s);
require(signer == owner, "ERC20Permit: invalid signature");
require(signer != address(0), "ECDSA: invalid signature");
```

We can then increase the nonce and perform the approve

```
nonces[owner]++;
_approve(owner, spender, amount);

```

Draft Example from Open Zeppelin

```
abstract contract ERC20Permit is ERC20, IERC20Permit, EIP712 {
    using Counters for Counters.Counter;

    mapping(address => Counters.Counter) private _nonces;

    // solhint-disable-next-line var-name-mixedcase
    bytes32 private immutable _PERMIT_TYPEHASH =
        keccak256("Permit(address owner,address spender,uint256 value,
        uint256 nonce,uint256 deadline)");

    /**
     * @dev Initializes the {EIP712} domain separator using the `name` pa
     * and setting `version` to `"1"`.
     *
     * It's a good idea to use the same `name` that is defined
     * as the ERC20 token name.
     */
    constructor(string memory name) EIP712(name, "1") {}

    /**
     * @dev See {IERC20Permit-permit}.
     */
    function permit(
        address owner,
        address spender,
        uint256 value,
        uint256 deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) public virtual override {
        require(block.timestamp <= deadline, "ERC20Permit: expired deadli

        bytes32 structHash = keccak256(abi.encode(_PERMIT_TYPEHASH, owner,
        spender, value, _useNonce(owner), deadline));

        bytes32 hash = _hashTypedDataV4(structHash);

        address signer = ECDSA.recover(hash, v, r, s);
        require(signer == owner, "ERC20Permit: invalid signature");

        _approve(owner, spender, value);
    }


```

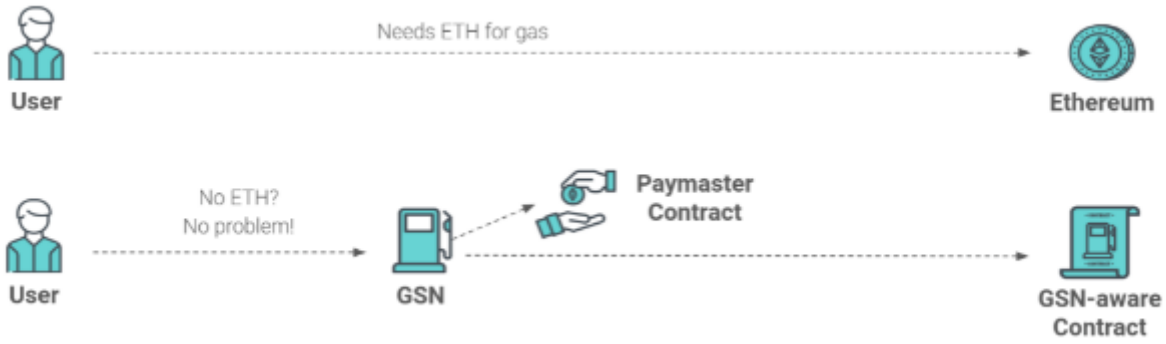
This is a step towards gassless transactions.

Meta Transactions

See Open Zeppelin meta transactions
(https://docs.openzeppelin.com/contracts/4.x/api/metatx#ERC2771Context)

Gas Station Network

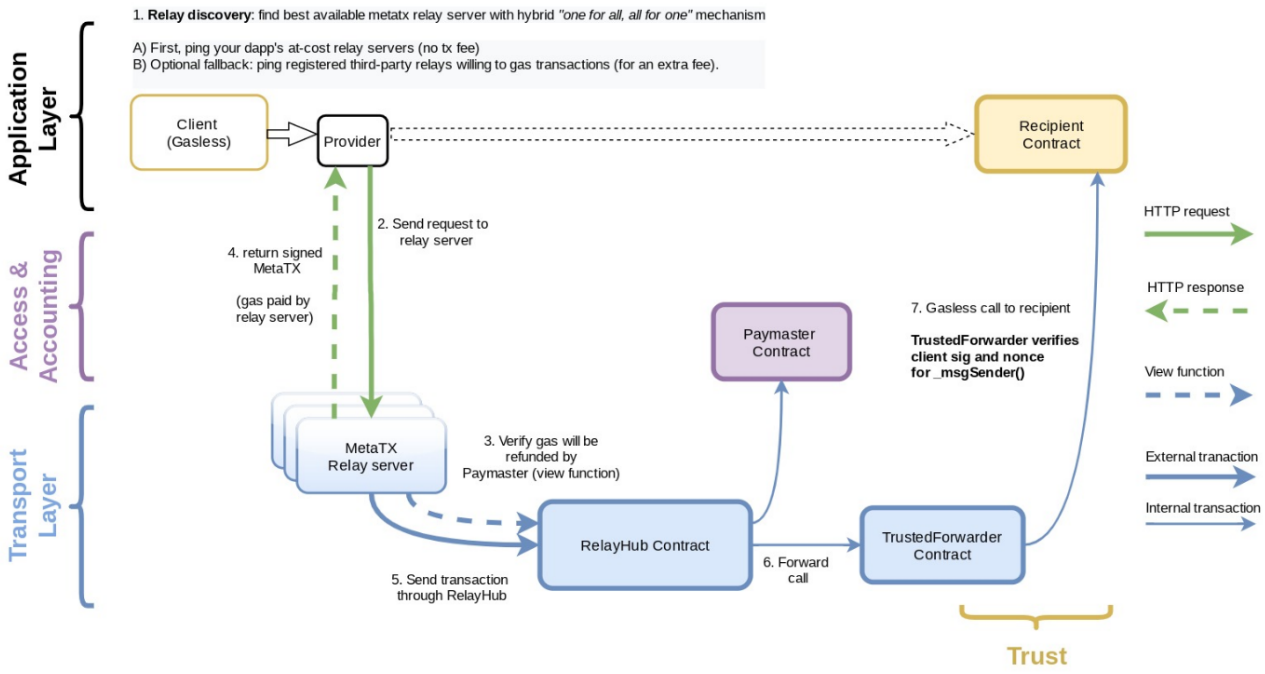
GSN1 is deprected in favour of Open GSN (https://opengsn.org/)
We will follow the tutorial (https://docs.opengsn.org/javascript-client/tutorial.html) from them



Use cases

- Pay gas in any token: Allow users to pay for gas in any token
- Pay gas in fiat: Allow users to pay for gas in fiat without having to go through KYC
- Privacy: Enabling ETH-less withdrawal of tokens sent to stealth addresses
- Onboarding: Allow dapps to subsidize the onboarding process for new users

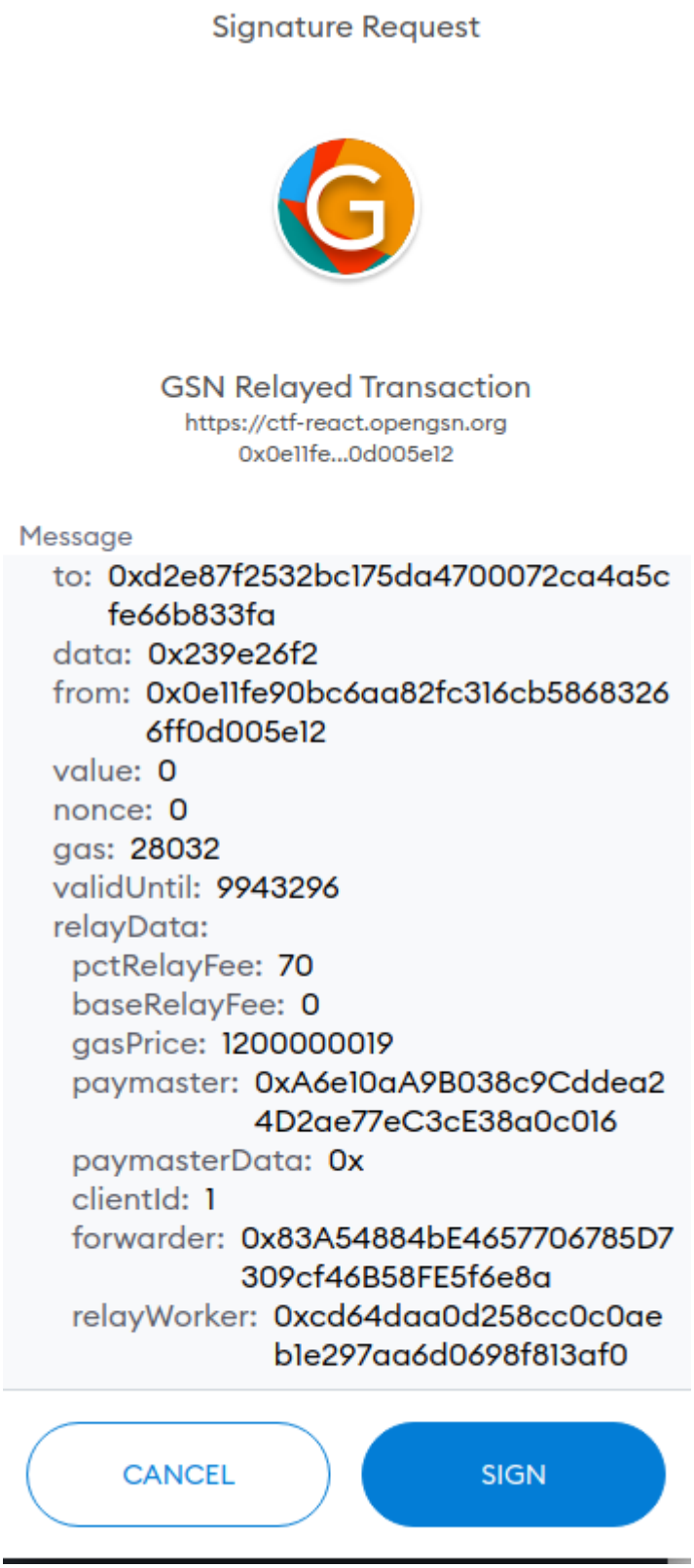
Architecture



- Client: signs & sends meta transaction to relay server
- Relay servers - provided by projects to give redundancy
- Paymaster: agrees to refund relay server for gas fees

    - The paymaster will have logic to decide whether to fund the transaction or not
    - If the paymaster decides to fund the transaction it pays for the transaction with its own ETH
- Trusted Forwarder: verifies sender signature and nonce
    - A simple security contract that verifies the accound details of the client
- Recipient contract: sees original sender (as _msgsender())
    - This contract will inherit from BaseRelayRecipient (https://github.com/opengsn/gsn/blob/release/contracts/BaseRelayRecipient.sol)
- RelayHub: connecting participants trustlessly

Metamask Details



Encode private network details

RPC address : http://18.130.233.246:9454 (http://18.130.233.246:9454)

ChainID : 20200520

Extropy Coin Address : 0x65e770F49625273e41f4B6790CE1105aC142f8a9