

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
<code>constructor</code>	0.5.0	Constructors must now be defined using the <code>constructor</code> keyword.
<code>visibility</code>	0.5.0	Explicit function visibility is now mandatory, defaults to <code>public</code> .
<code>abstract</code>	0.6.0	The keyword <code>abstract</code> has to be used if a contract does not implement all its functions.
<code>virtual</code>	0.6.0	Functions without implementation outside an interface have to be marked <code>virtual</code> .
<code>override</code>	0.6.0	When overriding a function or modifier, the new keyword <code>override</code> must be used.
<code>dotsyntax</code>	0.7.0	The following syntax is deprecated: <code>f.gas(...)</code> , <code>f.value(...)</code> and <code>(new C).value(...)</code> . Replace these calls by <code>f{gas: ..., value: ...}()</code> and <code>(new C){value: ...}()</code> .
<code>now</code>	0.7.0	The <code>now</code> keyword is deprecated. Use <code>block.timestamp</code> instead.
<code>constructor-visibility</code>	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 variable 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.
`2a5bc342ed616b5ba5732269001d3f1ef827552ae1114027bd3ecf1f086ba0f9`
`Address = 0x001d3f1ef827552ae1114027bd3ecf1f086ba0f9`

Contract Addresses

CREATE gives the address that a contract will be deployed to

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

CREATE2 introduced 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`.

```
1 // SPDX-License-Identifier: GPL-3.0
2 pragma solidity >=0.5.14 <0.9.0;
3
4 library L {
5     function f(uint256) external {}
6 }
7
8 contract C {
9     function g() public pure returns (bytes4) {
10         return L.f.selector;
11     }
12 }
```

Encoding the function signatures and parameters

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

```
1 pragma solidity ^0.8.0;
2
3 contract MyContract {
4
5     Foo otherContract;
6
7
8     function callOtherContract() public view returns (bool){
9         bool answer = otherContract.baz(69,true);
10        return answer;
11    }
12 }
13
14
15 contract Foo {
16     function bar(bytes3[2] memory) public pure {}
17     function baz(uint32 x, bool y) public pure returns (bool r) {
18         r = x > 32 || y;
19     }
20     function sam(bytes memory, bool, uint[] memory) public pure {}
21 }
```

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

0x0045:

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

0x0001:

In total

**0xcdcd77c000
00000000000000000000000045000000000000000000000000000000000000
0001**

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

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

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

```
1 | bytes memory payload =  
2 | abi.encodeWithSignature("baz(uint32, boolean)", 69, true);  
3 |  
4 | (bool success, bytes memory returnData) =  
5 | address(contractAddress).call(payload);  
6 |  
7 | 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`.

```
1 using ECDSA for bytes32;
2
3 function _verify(bytes32 data, bytes memory signature, address account)
4 internal pure returns (bool) {
5     return data
6         .toEthSignedMessageHash()
7         .recover(signature) == account;
8 }
```

EIP712 and EIP2612

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

Domain separator (for an ERC-20)

```
1 bytes32 eip712DomainHash = keccak256(
2     abi.encode(
3         keccak256(
4             "EIP712Domain(string name,string version,
5                 uint256 chainId,address verifyingContract)"
6         ),
7         keccak256(bytes(name())), // ERC-20 Name
8         keccak256(bytes("1")),    // Version
9         chainid(),
10        address(this)
11    )
12 );
```

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

```
1 function permit(address owner, address spender,
2     uint value, uint deadline, uint8 v, bytes32 r, bytes32 s) external
3 function nonces(address owner) external view returns (uint)
4 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

```
1 bytes32 hashStruct = keccak256(  
2     abi.encode(  
3         keccak256("Permit(address owner,address spender,  
4             uint256 value,uint256 nonce,uint256 deadline)"),  
5         owner,  
6         spender,  
7         amount,  
8         nonces[owner],  
9         deadline  
10    )  
11 );
```

This struct will ensure that the signature can only be 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

```
1 | bytes32 hash = keccak256(  
2 |     abi.encodePacked(uint16(0x1901), eip712DomainHash, hashStruct)  
3 | );
```

On receiving the signature we can verify with

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

We can then increase the nonce and perform the approve

```
1 | nonces[owner]++;  
2 | _approve(owner, spender, amount);  
3 |
```

Draft Example from Open Zeppelin

```

1  abstract contract ERC20Permit is ERC20, IERC20Permit, EIP712 {
2      using Counters for Counters.Counter;
3
4      mapping(address => Counters.Counter) private _nonces;
5
6      // solhint-disable-next-line var-name-mixedcase
7      bytes32 private immutable _PERMIT_TYPEHASH =
8          keccak256("Permit(address owner,address spender,uint256 value,
9              uint256 nonce,uint256 deadline)");
10
11     /**
12      * @dev Initializes the {EIP712} domain separator using the `name` pai
13      * and setting `version` to `"1"`.
14      *
15      * It's a good idea to use the same `name` that is defined
16      * as the ERC20 token name.
17      */
18     constructor(string memory name) EIP712(name, "1") {}
19
20     /**
21      * @dev See {IERC20Permit-permit}.
22      */
23     function permit(
24         address owner,
25         address spender,
26         uint256 value,
27         uint256 deadline,
28         uint8 v,
29         bytes32 r,
30         bytes32 s
31     ) public virtual override {
32         require(block.timestamp <= deadline, "ERC20Permit: expired deadli
33
34         bytes32 structHash = keccak256(abi.encode(_PERMIT_TYPEHASH, owner,
35             spender, value, _useNonce(owner), deadline));
36
37         bytes32 hash = _hashTypedDataV4(structHash);
38
39         address signer = ECDSA.recover(hash, v, r, s);
40         require(signer == owner, "ERC20Permit: invalid signature");
41
42         _approve(owner, spender, value);
43     }
44
45

```

This is a step towards gassless transactions.

Meta Transactions

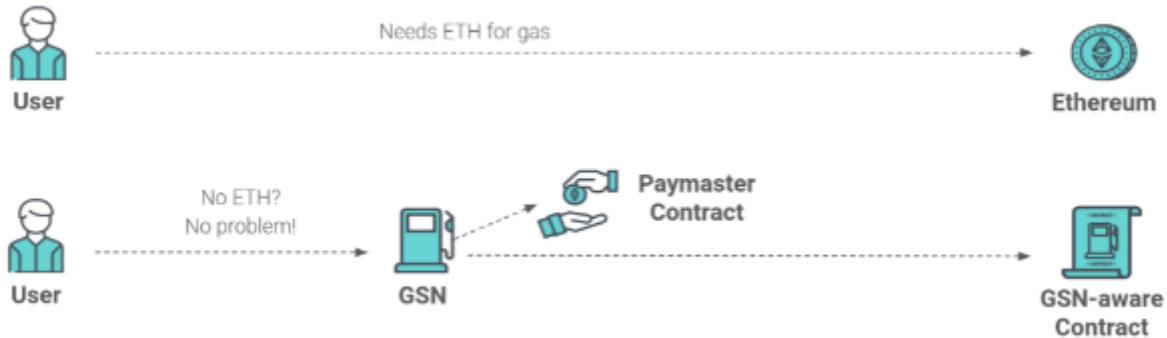
See Open Zeppelin meta transactions

(<https://docs.openzeppelin.com/contracts/4.x/api/metatx#ERC2771Context>)

Gas Station Network

GSN1 is deprecated in favour of Open GSN (<https://opengsn.org/>)

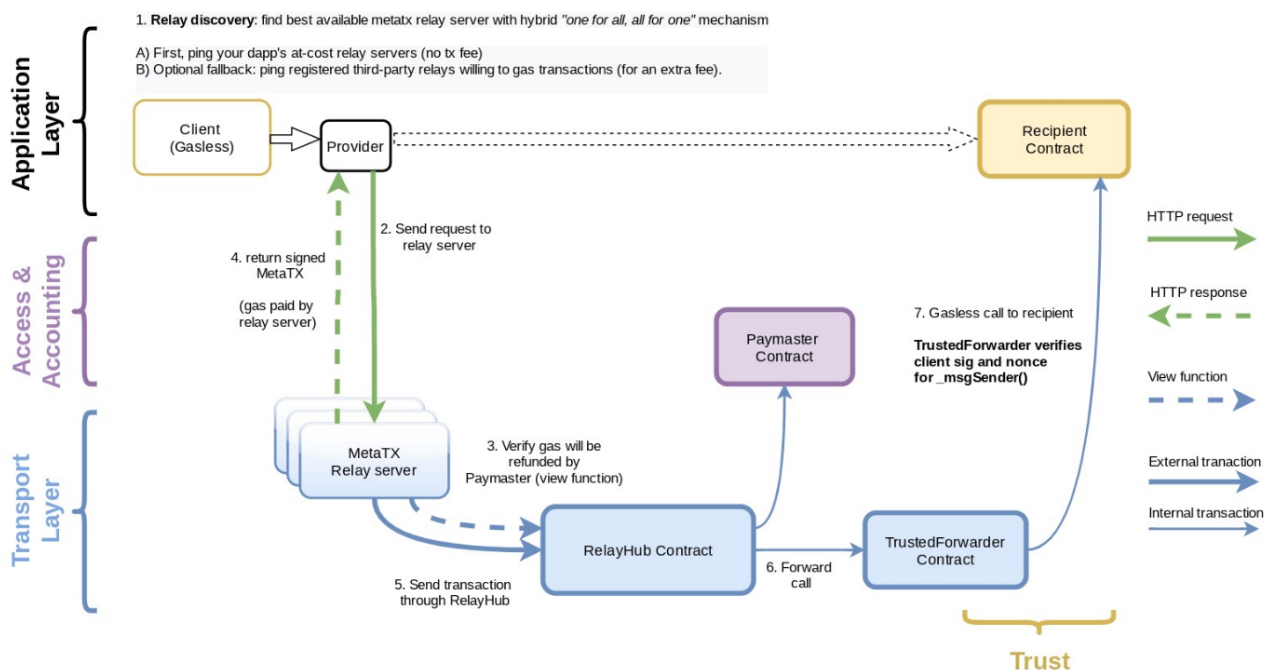
We will follow the tutorial (<https://docs.opengsn.org/javascript-client/tutorial.html>) from them



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 account 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

Polygon and Biconomy

See Polygon documentation (<https://docs.polygon.technology/docs/develop/metatransactions/getting-started/>)

See Biconomy (<https://docs.biconomy.io/tutorials/native-meta-transactions>) for native meta transactions

Metamask Details

Signature Request



GSN Relayed Transaction

<https://ctf-react.opengsn.org>
0x0e11fe...0d005e12

Message

```
to: 0xd2e87f2532bc175da4700072ca4a5c
    fe66b833fa
data: 0x239e26f2
from: 0x0e11fe90bc6aa82fc316cb5868326
    6ff0d005e12
value: 0
nonce: 0
gas: 28032
validUntil: 9943296
relayData:
  pctRelayFee: 70
  baseRelayFee: 0
  gasPrice: 1200000019
  paymaster: 0xA6e10aA9B038c9Cddea2
    4D2ae77eC3cE38a0c016
  paymasterData: 0x
  clientId: 1
  forwarder: 0x83A54884bE4657706785D7
    309cf46B58FE5f6e8a
  relayWorker: 0xcd64daa0d258cc0c0ae
    b1e297aa6d0698f813af0
```

CANCEL

SIGN

Encode private network details

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

ChainID : 20200520

Extropy Coin Address : 0x65e770F49625273e41f4B6790CE1105aC142f8a9

