Blockchain

- Solidity

Solidity

This page will cover setting up a local environment for working with Solidity. This is not required to deploy an ERC20 token, but it is a good learning experience to set up these environments yourself instead of using a provided service like Remix. That said, Remix is a *really* neat tool, and I would recommend getting into some basic solidity there first and deploying a test contract to ropsten or some other etereum testnet. For that process, I found *a lot* of information available online, so I won't cover it here, but here's my ERC20 token GitHub repository. The code there may be more up-to-date then the code on this page, so there will probably be discrepancies from the screenshots and snippets seen here as I work on the repository.

Once you deploy on Remix, if you still want to get a development environment and repository of your own setup, this page might be useful to you.

ERC20 Token Standard

Solidity Documentation

OpenZeppelin Documentation **This is a great resource**

OpenZeppelin ERC20 Decoumentation specific to the `IERC20` interface I inherit in my token contract.

OpenZeppelin Contracts GitHub Useful for looking at how things are implemented

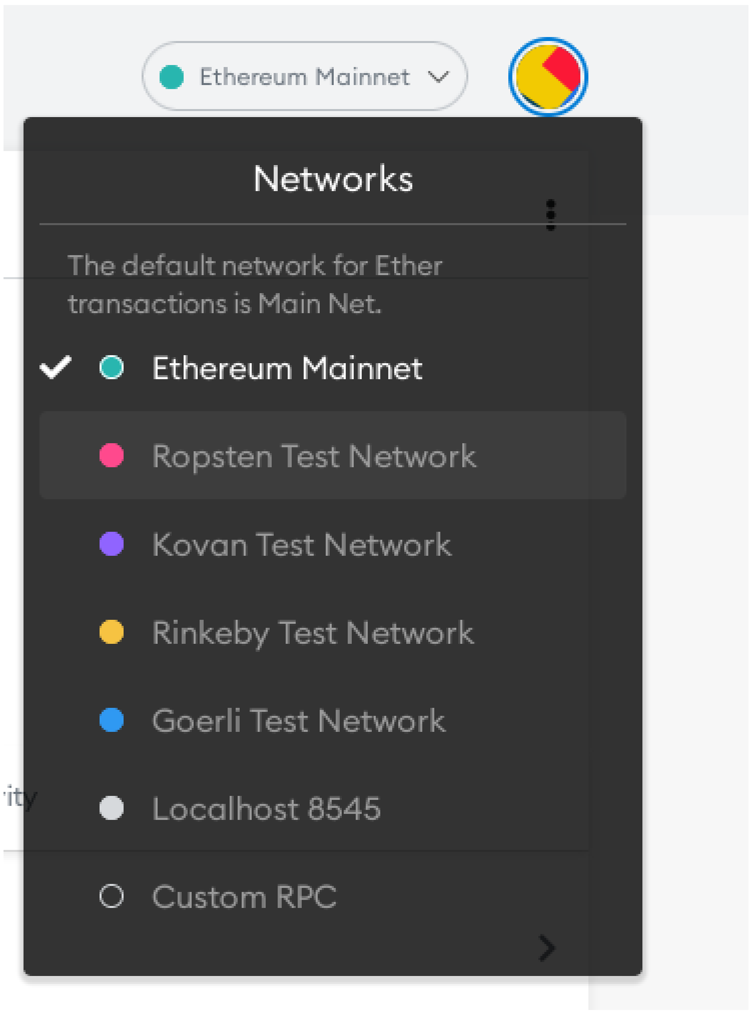
Remix ETH Web IDE **You can use this Web IDE as an alternative to setting up a local development environment.**

MetaMask

You will need the MetaMask browser extension and wallet. Once you install it onto your web browser, **be sure to save your mnemonic phrase.** This goes for all wallets, but we will specifically need this phrase to deploy our contract.

Switch to the Ropsten testnet within metamask, and visit the link below and enter your wallet address to *give yourself free test ethereum*. This is only for testing on the Ropsten network.

Ropsten ETH Faucet for testnet use only



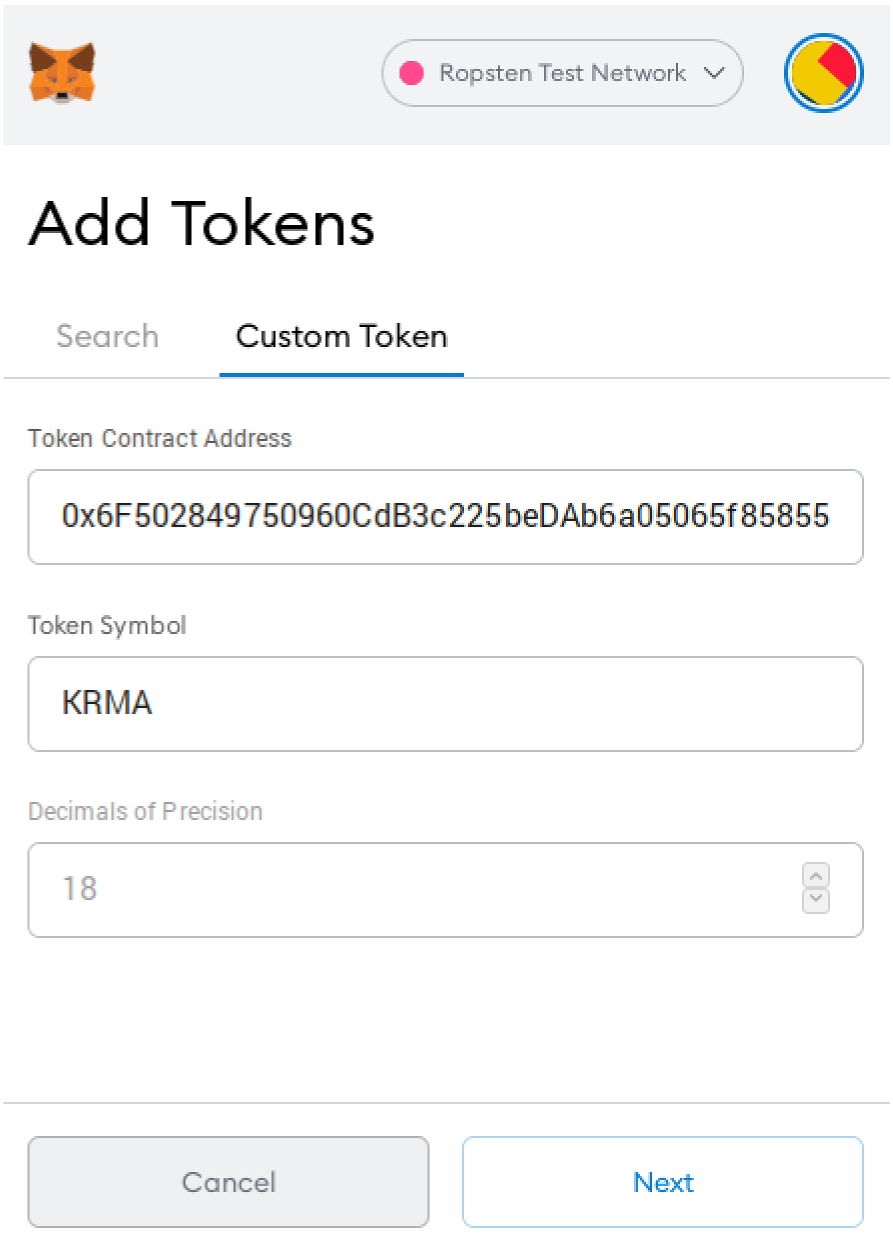
> You will need this test ETH to pay the transaction fees for deploying your contract when creating your ERC20 token. Don't skip this step, visit the link to the ropsten faucet above.

Custom Coins

You will only need to do this once you have your contract deployed

To get your custom token balances to show in metamask, you need to add a custom coin. Be sure you are on ropsten, or your testnet of choice, and see the images below.

In your MetaMask wallet, click `Add Token`, and go to the `Custom Token` tab. Enter your contract address you recieved when deploying.



Add Tokens

Token Contract Address

0x6F502849750960CdB3c225beDAb6a05065f85855

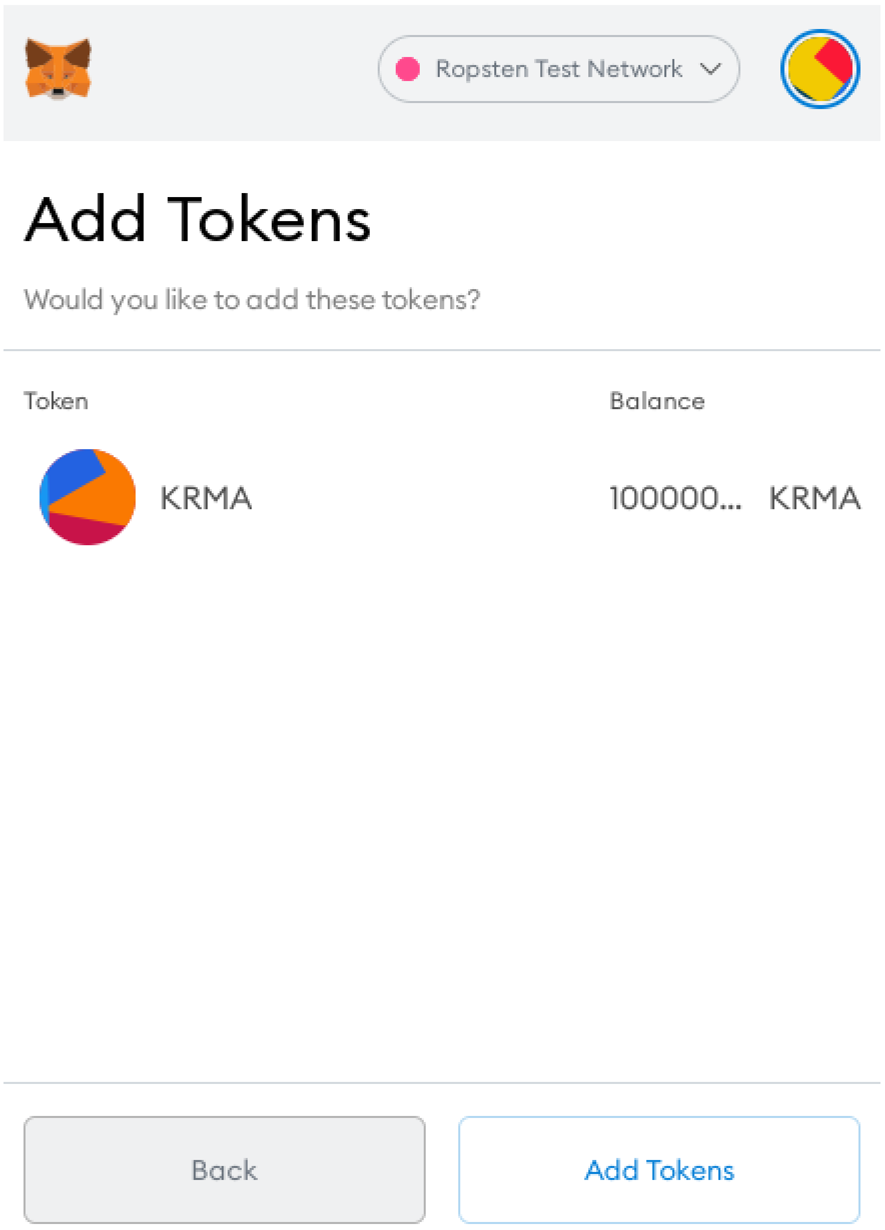
Token Symbol

KRMA

Decimals of Precision

18

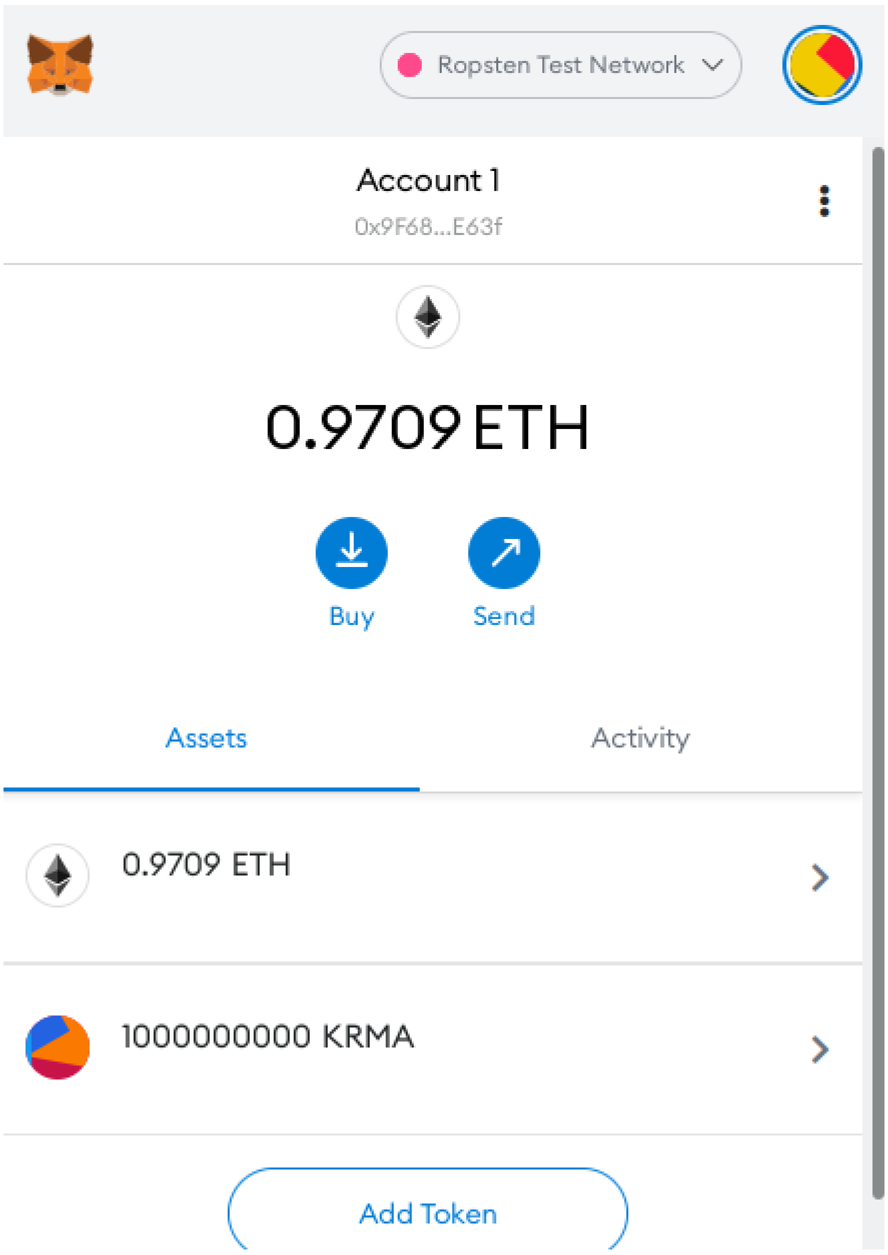
Click `Add Tokens` on the confimation screen, and they will appear in your wallet!



Add Tokens

Would you like to add these tokens?

Token	Balance
KRMA	100000... KRMA



Compiling Solidity With Truffle

Truffle Documentation

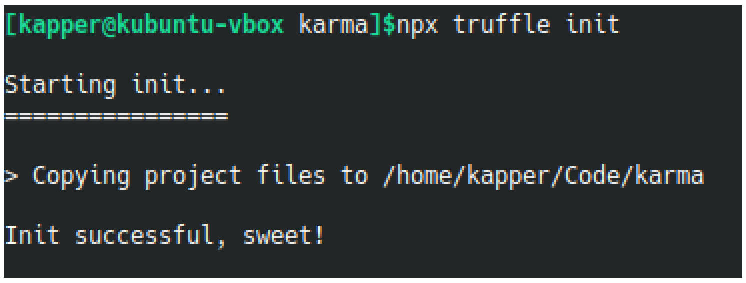
We need to install the Truffle compiler -

```
npm install --save-dev truffle
```

> The next steps will overwrite your `contracts/` and `test/` directories. If you have files there you want to keep, move or copy them somewhere else. After running the following initialization command, move them back, but keep any other additional files that Truffle has generated.

Run the following command to initialize a new Truffle project

```
npx truffle init
```



```
[kapper@kubuntu-vbox karma]$npx truffle init

Starting init...
================

> Copying project files to /home/kapper/Code/karma

Init successful, sweet!
```

Check the `truffle-config.js` generated by `npx truffle init`, and modify your compiler version as needed. For me, all active settings in this config are seen below. This will work for compiling locally, but not on any remote public network. The next section will cover deploying to a remote network.

```
// truffle-config.js
module.exports = {
  // Set default mocha options here, use special reporters etc.
  mocha: {
  },
  // Configure your compilers
  compilers: {
    solc: {
      version: "0.8.0",    // <------- Modify your version if needed!
    }
  },
  db: {
    enabled: false
  }
};
```

Project Setup

Solidity is the language I used to create an ERC20 Token. This section will cover compilation of a contract on a local machine. If it compiles locally, it will deploy successfully. The sections after this will cover deploying Solidity projects to a remote networks with Truffle, and then deploying upgradeable ERC20 contracts. After that deploy is done, your token will be available on the Ropsten test net, and can be sent between wallets or added to Pancakeswap or a similar service.

Install Solidity Compiler

How to setup an NPM project

How to setup a Solidity project

First, we need to make a project directory and install everything we need. If you don't have `git` or `npm`, you'll need to install them first. We will also install the Truffle compiler -

```
mkdir /some/project/dir
cd /some/project/dir
git init	// Git is useful. Use it as needed, or don't. I won't cover any git commands here.
npm init -y		// Modify the generated package.json file, if needed
```

if you want to use OpenZeppelin, you will need to run the following command to install it, making it available to import in your contracts

```
npm install --save-dev @openzeppelin/contracts
```

Now, within a new `contracts/contract-ERC20.sol` file we can import the ETH ERC20 interface provided by OpenZeppelin, define a new contract `Karma`, and provide our own definitions for the member functions declared in the `IERC20` interface -

```
// Copyright [2021] - [2021], [Shaun Reed] and [Karma] contributors
// SPDX-License-Identifier: MIT

pragma solidity >= 0.8.0;

// ----------------------------------------------------------------------------
// Import ERC Token Standard #20 Interface
//   ETH EIP repo: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md
// ----------------------------------------------------------------------------
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";

// ----------------------------------------------------------------------------
// SourceCoin Contract
// ----------------------------------------------------------------------------
contract Karma is IERC20
{
    string public name;
    string public symbol;
    uint8 public decimals; // 18 decimals is the strongly suggested default, avoid changing it
    uint256 public _totalSupply;

    // Balances for each account; A hashmap using wallet address as key and uint as value
    mapping(address => uint) balances;
```

```
// Owner of account approves the transfer of an amount to another account
mapping(address => mapping(address => uint)) allowed;



/**
 * Constrctor function
 *
 * Initializes contract with initial supply tokens to the creator of the contract
 */
constructor()
{
    name = "Karma";    // Name of the token
    symbol = "KRMA";         // Abbreviation of the token
    decimals = 18;          // Number of decimals that can be used to split token



    // FORMAT: <SUPPLY><DECIMALS>
    // Where SUPPLY is the number of coins in base 10 decimal notation
    // And DECIMALS is a trailing number of 0's; Count must match `decimals` value set above
    // 1000 000 000 000000000000000000 == 1 billion total supply;
    //  + trailing 0's represent the 18 decimal locations that can be used to send fractions
    _totalSupply = 1000000000000000000000000000;



    // Set the remaining balance of the contract owner to the total supply
    balances[msg.sender] = _totalSupply; // msg.sender is the calling address for this constructor
    // Transfer the total supply to the contract owner on initialization
    emit Transfer(address(0), msg.sender, _totalSupply); // address(0) is used to represent a new TX
}



// Get the total circulating supply of the token
function totalSupply() public override view returns (uint)
{
    // By subtracting from tokens held at address(0), we provide an address to 'burn' the supply
    return _totalSupply - balances[address(0)]; // Subtract from tokens held at address(0)
}

// Get the token balance for account `tokenOwner`
function balanceOf(address tokenOwner) public override view returns (uint balance)
```

```
{
    return balances[tokenOwner]; // Return the balance of the owner's address
}

/// @param tokenOwner The address of the account owning tokens
/// @param spender The address of the account able to transfer the tokens
/// returns Amount of remaining tokens allowed to spent
function allowance(address tokenOwner, address spender) public override view returns (uint remaining)
{
    return allowed[tokenOwner][spender];
}

// Allow `spender` to withdraw from your account, multiple times, up to the `tokens`
// If this function is called again it overwrites the current allowance with _value.
function approve(address spender, uint tokens) public override returns (bool success)
{
    allowed[msg.sender][spender] = tokens;
    emit Approval(msg.sender, spender, tokens);
    return true;
}

// Transfer the balance from owner's account to another account
function transfer(address to, uint tokens) public override returns (bool success)
{
    balances[msg.sender] = balances[msg.sender] - tokens;
    balances[to] = balances[to] + tokens;
    emit Transfer(msg.sender, to, tokens);
    return true;
}

// Send `tokens` amount of tokens from address `from` to address `to`
// The transferFrom method is used for a withdraw workflow, allowing contracts to send
// tokens on your behalf, for example to "deposit" to a contract address and/or to charge
// fees in sub-currencies; the command should fail unless the _from account has
// deliberately authorized the sender of the message via some mechanism; we propose
// these standardized APIs for approval:
function transferFrom(address from, address to, uint tokens) public override returns (bool success)
{
    balances[from] = balances[from] - tokens;
    allowed[from][msg.sender] = allowed[from][msg.sender] - tokens;
```

```
        balances[to] = balances[to] - tokens;
        emit Transfer(from, to, tokens);
        return true;
    }


}
```

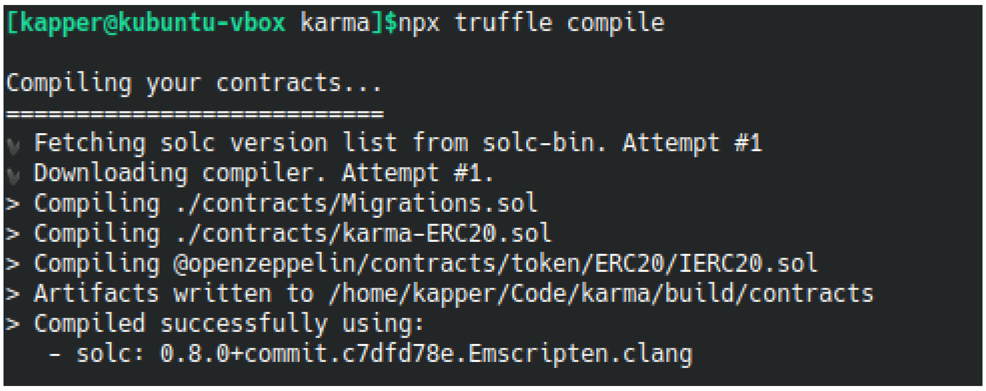
The next section will cover compiling this contract with Truffle.

Compiling Solidity With Truffle

Be sure to move any contracts you have into the `contracts/` directory, and any tests you have into the `test/` directory. For my project, I have a simple network test and a single contract. For an example ERC20, check shaunrd0/karma The screenshot below of my compilation output should provide some insight into how my project is structured.

Now, when we run the following command truffle will compile all contracts in the `contracts/` directory.

```
npx truffle compile
```


```
[kapper@kubuntu-vbox karma]$npx truffle compile

Compiling your contracts...
===========================
✓ Fetching solc version list from solc-bin. Attempt #1
✓ Downloading compiler. Attempt #1.
> Compiling ./contracts/Migrations.sol
> Compiling ./contracts/karma-ERC20.sol
> Compiling @openzeppelin/contracts/token/ERC20/IERC20.sol
> Artifacts written to /home/kapper/Code/karma/build/contracts
> Compiled successfully using:
   - solc: 0.8.0+commit.c7dfd78e.Emscripten.clang
```

Now that our contract compiles, we need to configure truffle to connect to a testnet so we can deploy to the public test network. The next section will cover doing this.

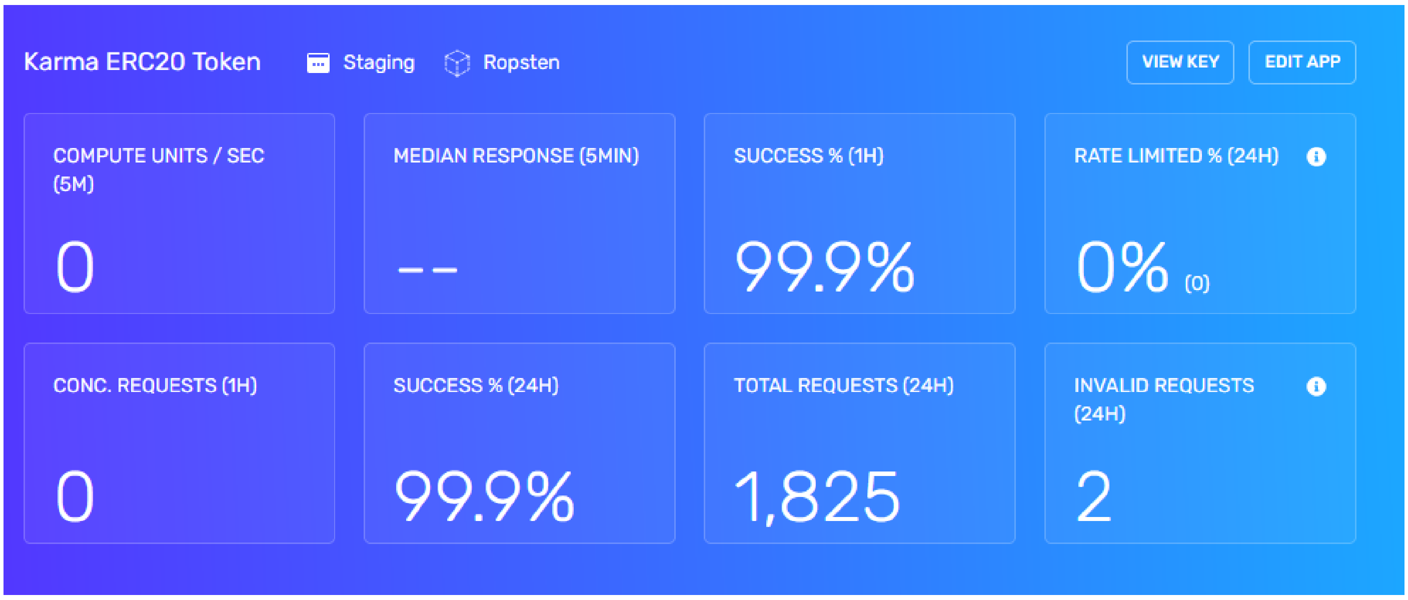
Deploy Truffle to Public Networks

Deploying Truffle to Remote Networks

Install `hdwallet-provider` for Truffle. We will use this to configure our wallet address when deploying the contract.

```
npm install --save-dev @truffle/hdwallet-provider
```

Grab your alchemy API key first, because you will need it! Once registered, the Alchemy dashboard will let you create a new project. Name it whatever you want, put it on whatever network you want **just make sure this matches the network you put in your config below**. Once you create the app, click `View Key` in the screenshot below. This can be found on the dashboard for the application itself. Copy the HTTPS key provided.



COMPUTE UNITS / SEC (5M) 0
MEDIAN RESPONSE (5MIN) --
SUCCESS % (1H) 99.9%
RATE LIMITED % (24H) 0% (0)
CONC. REQUESTS (1H) 0
SUCCESS % (24H) 99.9%
TOTAL REQUESTS (24H) 1,825
INVALID REQUESTS (24H) 2

Now you're ready to look at your config. Add your mnemonic attached to your MetaMask wallet as the `phrase`, and add your Alchemy API key as the `providerOrUrl` -

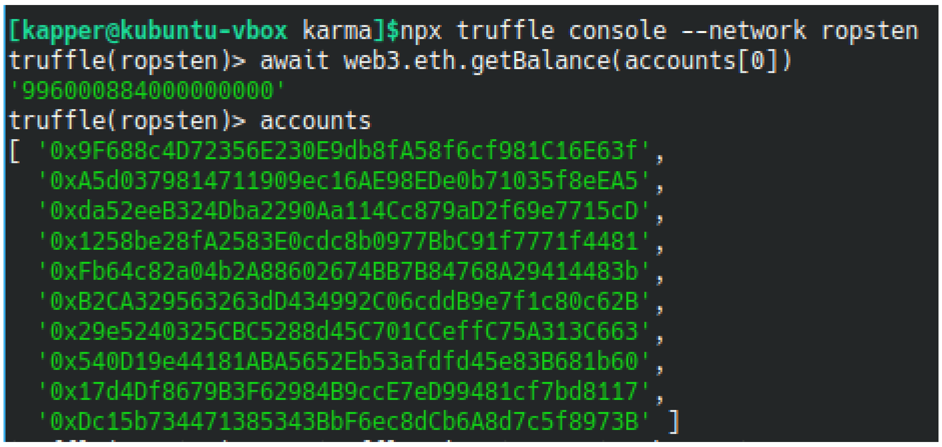
> Do not commit or post the raw contents of this file, unless you use a `secrets.json` or some other method to abstract your personal details. You never want to share your mnemonic phrase, or your alchemy API key with anyone.

```
// truffle-config.js
const HDWalletProvider = require('@truffle/hdwallet-provider');

module.exports = {
  networks: {
    ropsten: {
      provider: () => new HDWalletProvider({
        mnemonic: {
          phrase: 'word word word word word word word word word word word word word word'
        },
        providerOrUrl: "https://eth-ropsten.alchemyapi.io/v2/xxxxxxx_YOUR_ALCHEMY_API_KEY_xxxxxxxx",
        chainId: 3▯// <--- Don't forget this! For me, excluding this line caused errors. cainId 3 is for ropsten
      }),
      network_id: 3,       // Ropsten's id
      gas: 5500000,        // Ropsten has a lower block limit than mainnet
      confirmations: 2,    // # of confs to wait between deployments. (default: 0)
      timeoutBlocks: 200,  // # of blocks before a deployment times out  (minimum/default: 50)
      skipDryRun: true     // Skip dry run before migrations? (default: false for public nets )
    },
  },
  // Set default mocha options here, use special reporters etc.
  mocha: {
  },

  // Configure your compilers
  compilers: {
    solc: {
      version: "0.8.0",    // <------- Modify your version if needed!
    }
  },
  db: {
    enabled: false
  }
};
```

Now we are ready to test that we can connect to the ropsten network, and verify our wallet connection at the same time.

```
npx truffle console --network ropsten
```

This should open a console prompt, type `await web3.eth.getBalance(accounts[0])`, followed by `accounts`. If you get a response, things are configured correctly! You should notice your wallet address is listed after entering the second `accounts` command.



```
[kapper@kubuntu-vbox karma]$npx truffle console --network ropsten
truffle(ropsten)> await web3.eth.getBalance(accounts[0])
'996000884000000000'
truffle(ropsten)> accounts
[ '0x9F688c4D72356E230E9db8fA58f6cf981C16E63f',
  '0xA5d0379814711909ec16AE98EDe0b71035f8eEA5',
  '0xda52eeB324Dba2290Aa114Cc879aD2f69e7715cD',
  '0x1258be28fA2583E0cdc8b0977BbC91f7771f4481',
  '0xFb64c82a04b2A88602674BB7B84768A29414483b',
  '0xB2CA329563263dD434992C06cddB9e7f1c80c62B',
  '0x29e5240325CBC5288d45C701CCeffC75A313C663',
  '0x540D19e44181ABA5652Eb53afdfd45e83B681b60',
  '0x17d4Df8679B3F62984B9ccE7eD99481cf7bd8117',
  '0xDc15b734471385343BbF6ec8dCb6A8d7c5f8973B' ]
```

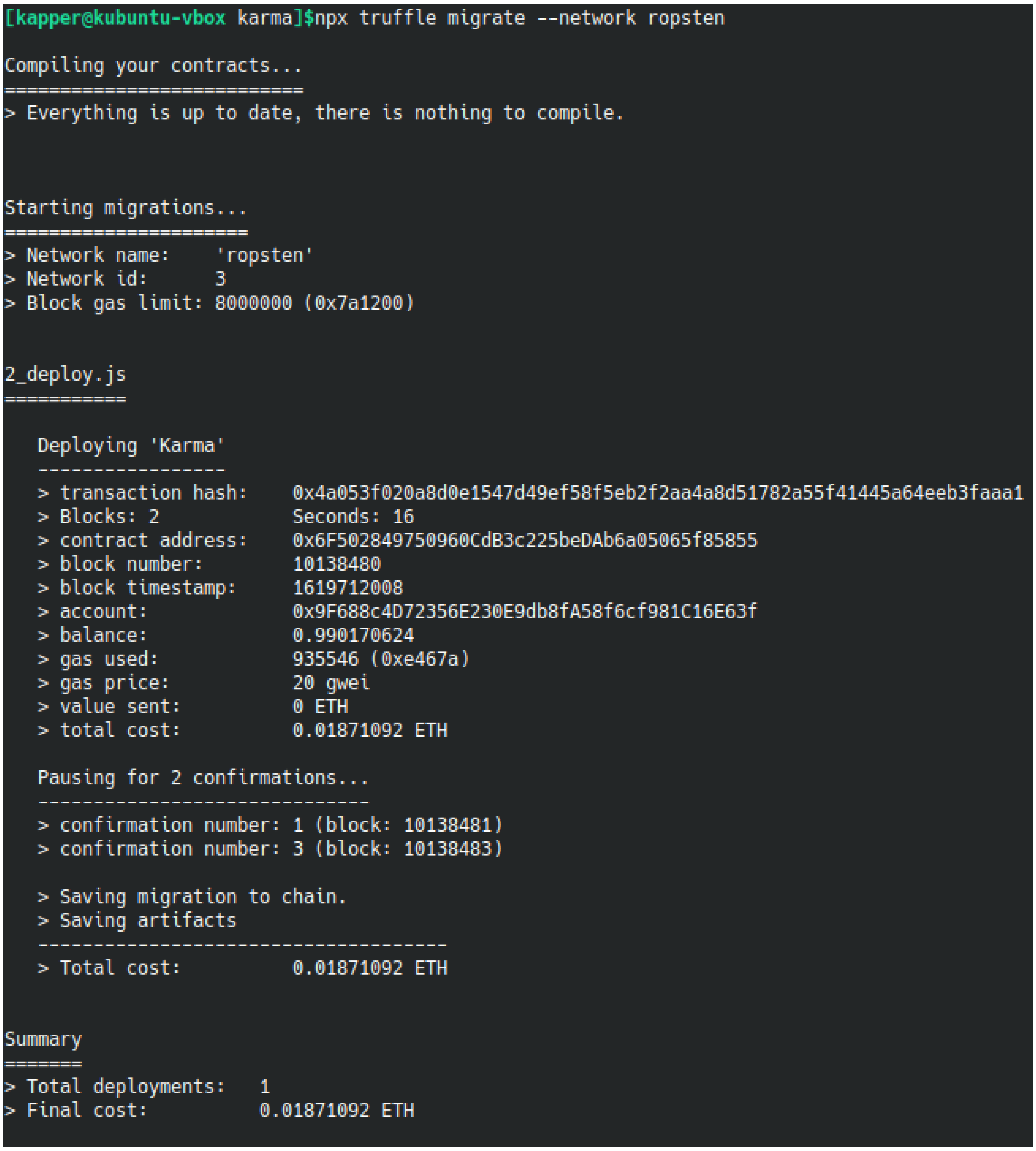
One final step before we can deploy our contract to ropsten. Create the file `migrations/2_deploy.js`, and add the contents below. Be sure to pay attention to comments and change the values as needed.

```
// migrations/2_deploy.js
// Set variable name Karma to your whatever you named your token
const Karma = artifacts.require("Karma"); // <-- Set string in call to articfacts.require to your token name


module.exports = function (deployer) {
   deployer.deploy(Karma); // <----- Use const variable declared above in call to deployer.deploy
};
```

Now, we are finally ready to deploy our contract. run this final command to deploy to ropsten testnet.

```
npx truffle migrate --network ropsten
```

This may take a while, but when it completes you should see the output below



```
[kapper@kubuntu-vbox karma]$npx truffle migrate --network ropsten

Compiling your contracts...
===========================
> Everything is up to date, there is nothing to compile.



Starting migrations...
======================
> Network name:    'ropsten'
> Network id:      3
> Block gas limit: 8000000 (0x7a1200)


2_deploy.js
===========

   Deploying 'Karma'
   -----------------
   > transaction hash:    0x4a053f020a8d0e1547d49ef58f5eb2f2aa4a8d51782a55f41445a64eeb3faaa1
   > Blocks: 2            Seconds: 16
   > contract address:    0x6F502849750960CdB3c225beDAb6a05065f85855
   > block number:        10138480
   > block timestamp:     1619712008
   > account:             0x9F688c4D72356E230E9db8fA58f6cf981C16E63f
   > balance:             0.990170624
   > gas used:            935546 (0xe467a)
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.01871092 ETH

   Pausing for 2 confirmations...
   ------------------------------
   > confirmation number: 1 (block: 10138481)
   > confirmation number: 3 (block: 10138483)

   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:          0.01871092 ETH


Summary
=======
> Total deployments:   1
> Final cost:          0.01871092 ETH
```

We have deployed Karma to the ETH ropsten testnet! You can take the `contract address` and view it on EtherScan. Here's the link for the contract we deployed in this section. **All of my source code for this token's contract can also be found there, under the `contract` tab.** It is useful, once you have a contract deployed or some transactions to look at, to explore etherscan a bit to see how things are organized. Notice on my contract page, you can see that 1 Billion Karmas were minted and sent to my wallet address when the contract was deployed, since I am the contract owner, and our contrract constructor looks like this -

```
/**
 * Constrctor function
```

```
  *
  * Initializes contract with initial supply tokens to the creator of the contract
  */
 constructor()
 {
    name = "Karma";    // Name of the token
    symbol = "KRMA";         // Abbreviation of the token
    decimals = 18;          // Number of decimals that can be used to split token


    // FORMAT: <SUPPLY><DECIMALS>
    // Where SUPPLY is the number of coins in base 10 decimal notation
    // And DECIMALS is a trailing number of 0's; Count must match `decimals` value set above
    // 1000 000 000 000000000000000000 == 1 billion total supply;
    //  + trailing 0's represent the 18 decimal locations that can be used to send fractions
    _totalSupply = 1000000000000000000000000000;


    // Set the remaining balance of the contract owner to the total supply
    balances[msg.sender] = _totalSupply; // msg.sender is the calling address for this constructor
    // Transfer the total supply to the contract owner on initialization
    emit Transfer(address(0), msg.sender, _totalSupply); // address(0) is used to represent a new TX
 }
```

This constructor only happens wheen the contract is deployed. Currently, with the contract we have deployed, the only way to update the contract implementation is to effectively deploy a new contract and tell all clients to direct their calls to the new address on ropsten. This could be a simple or complicated process, depending on the use of your token.

If this just didn't seem like it would scale well for you, the next section will look into using a proxy pattern to deploy a `ProxyAdmin` which would own a `TransparentUpgradeableProxy` - this `TransparentUpgradeableProxy` points to a contract address on the ethereum network that implements the functionality of `Karma`, or whatever you have named your token. Laying out the project this way, things make sense and you provide yourself with the tools you need to upgrade contracts without any clients needing to redirect any of their calls to a new address.

A complete example of a non-upgradeable token can be found at shaunrd0/karma/basic-karma

Upgradeable ERC20

For deploying upgradeable contracts, we can use the same `truffle.config.js` as we did in the sections above.

Upgradeable Contracts For Truffle

First, we need to install `@openzeppelin/contracts-upgradeable` to your project by running the following command -

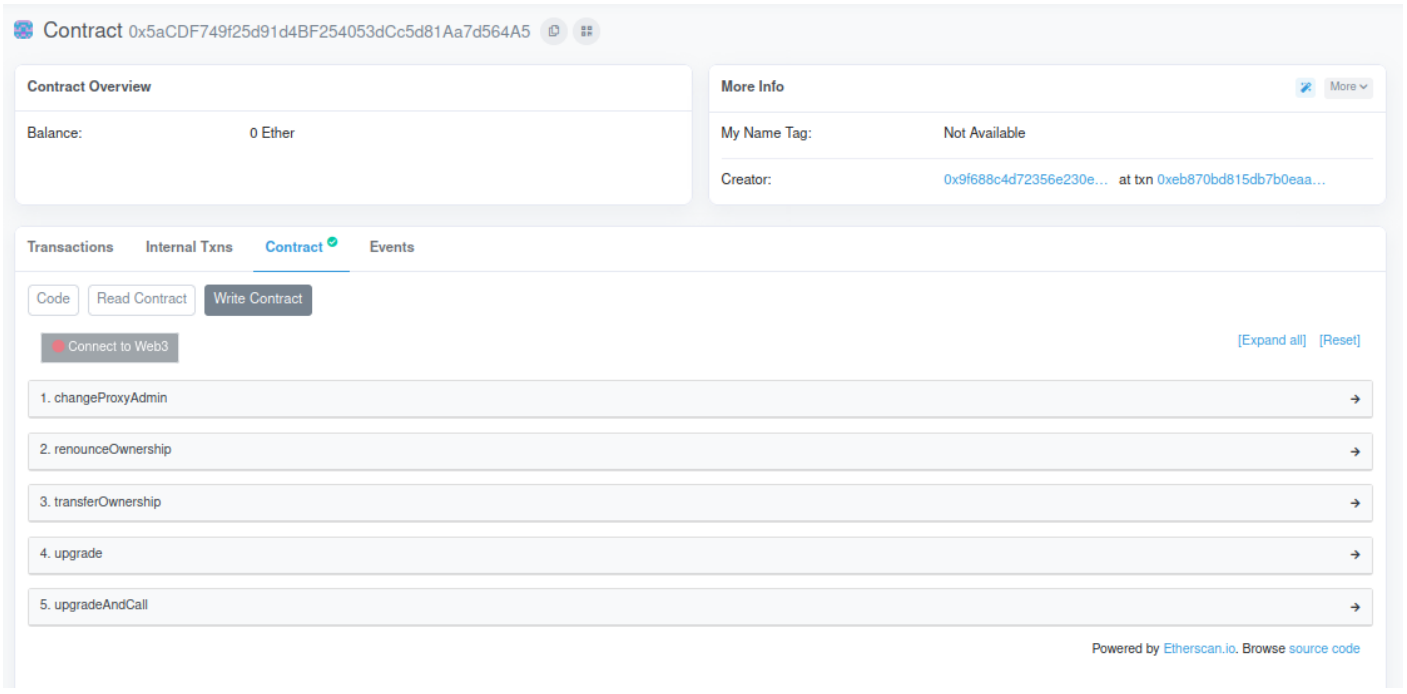
```
npm i --save-dev @openzeppelin/contracts-upgradeable
```

If this is your first time deploying using `@openzeppelin/truffle-upgrades`, you may notice an additional deploy of a `ProxyAdmin` that happens when you make your very first deploy. This `ProxyAdmin` is owned by your wallet address, and you can use it to manage the proxied contracts you deploy in the future. OpenZeppelin will automatically deploy **one** `ProxyAdmin` on each network you deploy on, and this deploy will only happen the first time you deploy on the network.

The only code we need to provide for this to happen is seen below in the first few lines of the `migrations/3_deploy_proxy.js` file.

```
const { deployProxy } = require('@openzeppelin/truffle-upgrades');
```

When we include this `deployProxy` and using it to deploy a proxy contract, OpenZeppelin handles making sure we have a `ProxyAdmin` deployed on the network first. We don't need to otherwise manually define the `ProxyAdmin` contract or deploy process. Here is my ProxyAdmin - you'll notice you have the following functions available in the `write contract` section on etherscan


Contract 0x5aCDF749f25d91d4BF254053dCc5d81Aa7d564A5

Contract Overview
Balance: 0 Ether

More Info
My Name Tag: Not Available
Creator: 0x9f688c4d72356e230e… at txn 0xeb870bd815db7b0eaa…

1. changeProxyAdmin
2. renounceOwnership
3. transferOwnership
4. upgrade
5. upgradeAndCall

If we click `Connect to Web3`, metamask will prompt us to choose an account to connect to. In order for this to work, **your metamask account must be on the same network as the contract**

you are viewing.

If you try to connect to my `ProxyAdmin`, you'll notice you lack permissions to run any of the functions in `Write Contract`. It would be the same for me if I tried to connect to your `ProxyAdmin`. This protects our contracts and allows only the owner of the `ProxyAdmin` to manage their own contracts.

For now, it is enough to know this contract exists, when it is created, and what it is for. Later on, we will use this contract to upgrade our proxy implementation.

A complete example of an upgradeable token can be found at shaunrd0/karma

Contracts

Let's say we have the following contract

```
// Copyright [2021] - [2021], [Shaun Reed] and [Karma] contributors
// SPDX-License-Identifier: MIT

pragma solidity >= 0.8.0;

// --------------------------------------------------------------------------
// Import ERC Token Standard #20 Interface
//   ETH EIP repo: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md
// --------------------------------------------------------------------------
import "@openzeppelin/contracts-upgradeable/token/ERC20/ERC20Upgradeable.sol";
import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";

// --------------------------------------------------------------------------
// Karma Contract
// --------------------------------------------------------------------------

contract Karma is Initializable, ERC20Upgradeable
{
    function initialize(string memory name, string memory symbol, uint256 initialSupply) public virtual initializer {
        __ERC20_init(name, symbol);
        _mint(_msgSender(), initialSupply);
    }
}
```

And we want to add an additional function, `isToken`, just to test that our upgrades are working. This function simply returns `true`, since `Karma` is a token. Notice the contract above does *not* define or declare this function.

To upgrade `Karma`, we add the function below, and rename the contract to `KarmaV2`

```
// Copyright [2021] - [2021], [Shaun Reed] and [Karma] contributors
// SPDX-License-Identifier: MIT

pragma solidity >= 0.8.0;

// ---------------------------------------------------------------------------
// Import ERC Token Standard #20 Interface
//   ETH EIP repo: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md
// ---------------------------------------------------------------------------
import "@openzeppelin/contracts-upgradeable/token/ERC20/ERC20Upgradeable.sol";
import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";

// ---------------------------------------------------------------------------
// Karma Contract
// ---------------------------------------------------------------------------

contract KarmaV2 is Initializable, ERC20Upgradeable
{
    function initialize(string memory name, string memory symbol, uint256 initialSupply) public virtual initializer {
        __ERC20_init(name, symbol);
        _mint(_msgSender(), initialSupply);
    }

    function isToken() public view returns (bool)
    {
        return true;
    }
}
```

Now we have defined all the contracts we need for an upgradeable ERC20 token, and we have written the contract for our upgrade. In the next section, we will define the needed Migrations to deploy the original `Karma` to ETH ropsten test network. Then, we will upgrade `Karma` to `KarmaV2`, and finally, to `KarmaV3`.

Migrations

We will need to create the following migrations, the `./migrations/` folder inside your project directory. Each file has a comment with the required name. It is important to prefix these migrations with `1_`, `2_`, `3_`, etc - as Truffle uses this order to track migration status and pick up where it left off should a subsequent migration be interrupted or fail.

```
// migrations/1_initial_migration.js
const Migrations = artifacts.require("Migrations");

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

```
// migrations/2_deploy_karma.js
const Karma = artifacts.require('Karma');

module.exports = async function (deployer) {
    await deployer.deploy(Karma);
};
```

```
// migrations/3_deploy_proxy.js
const Karma = artifacts.require('Karma');

const { deployProxy } = require('@openzeppelin/truffle-upgrades');

module.exports = async function (deployer) {
    await deployProxy(
        Karma,
        ['Karma', 'KRMA', '1000000000000000000000000000'],
        { deployer, initializer: 'initialize' }
    );
};
```

Deploying and Verifying

Now, having defined the above contracts and migrations, we are ready to deploy to a public test network. Run `npx truffle migrate --network ropsten` to deploy to ETH's ropsten testnet, or replace `ropsten` with the testnet of your choice. The output from my deploy can be seen below. Note that there is no deploy seen in the output for my `ProxyAdmin` since it already exists. An account can only use one `ProxyAdmin` on each network, but can own several proxies.

```
npx truffle migrate --network ropsten
Compiling your contracts...
===========================
> Everything is up to date, there is nothing to compile.
```

```
Starting migrations...
======================
> Network name:    'ropsten'
> Network id:      3
> Block gas limit: 8000000 (0x7a1200)



1_initial_migration.js
======================

   Deploying 'Migrations'
   ----------------------
   > transaction hash:    0x17286139c9b789dd5a92bb36a5d58a510285d170f28dec48813c3c1ed8218658
   > Blocks: 0            Seconds: 28
   > contract address:    0x8EfAf71d6126b2A0e76A319c92303B3E6Fdc521c
   > block number:        10184001
   > block timestamp:     1620312442
   > account:             0x9F688c4D72356E230E9db8fA58f6cf981C16E63f
   > balance:             1.62242806275
   > gas used:            245600 (0x3bf60)
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.004912 ETH

   Pausing for 2 confirmations...
   ------------------------------
   > confirmation number: 1 (block: 10184002)
   > confirmation number: 2 (block: 10184003)

   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:            0.004912 ETH



2_deploy_karma.js
=================
   Deploying 'Karma'
   -----------------
   > transaction hash:    0x8757dcfb9ada79802219fbb4ff4851d56044681cef0e141c35a3fc81c61340fd
   > Blocks: 1            Seconds: 8
   > contract address:    0xb2281089EBf1baB9d701f8697b1eCBaC3319e00F
   > block number:        10184008
   > block timestamp:     1620312531
   > account:             0x9F688c4D72356E230E9db8fA58f6cf981C16E63f
   > balance:             1.58983600275
   > gas used:            1583690 (0x182a4a)
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.0316738 ETH

   Pausing for 2 confirmations...
   ------------------------------
   > confirmation number: 1 (block: 10184009)
   > confirmation number: 2 (block: 10184010)

   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:           0.0316738 ETH


3_deploy_proxy.js
=================

   Replacing 'Karma'
   -----------------
   > transaction hash:    0xca0c617b7f9668ad431c8199fdbedec7e68d08ee42961c89bea3f0670d8b6607
   > Blocks: 0            Seconds: 8
   > contract address:    0x957684dC3De2b93154b2561c7bC96875306E39A0
   > block number:        10184013
   > block timestamp:     1620312610
   > account:             0x9F688c4D72356E230E9db8fA58f6cf981C16E63f
   > balance:             1.55758594275
   > gas used:            1583690 (0x182a4a)
   > gas price:           20 gwei
   > value sent:          0 ETH
> total cost:          0.0316738 ETH

Pausing for 2 confirmations...
------------------------------
> confirmation number: 1 (block: 10184014)
> confirmation number: 2 (block: 10184015)

Deploying 'ProxyAdmin'
----------------------
> transaction hash:    0xc2733e6480aeb1746e00f8951ac770b9c03d325fa5baf0570ddd13e9f1fefb2c
> Blocks: 2            Seconds: 24
> contract address:    0xC51D55CCDe8996993D05EBFd9Ad481A4A782B82B
> block number:        10184017
> block timestamp:     1620312651
> account:             0x9F688c4D72356E230E9db8fA58f6cf981C16E63f
> balance:             1.54790554275
> gas used:            484020 (0x762b4)
> gas price:           20 gwei
> value sent:          0 ETH
> total cost:          0.0096804 ETH

Pausing for 2 confirmations...
------------------------------
> confirmation number: 1 (block: 10184018)
> confirmation number: 2 (block: 10184019)

Deploying 'TransparentUpgradeableProxy'
---------------------------------------
> transaction hash:    0x763e6ef2c2828a320c46830f224140c2c30305ee6453eb7b0bd46d17538fede0
> Blocks: 1            Seconds: 5
> contract address:    0x438B6a24d3581c379F51Ae389bf37236ae94BEA8
> block number:        10184022
> block timestamp:     1620312693
> account:             0x9F688c4D72356E230E9db8fA58f6cf981C16E63f
> balance:             1.53359502275
> gas used:            715526 (0xaeb06)
> gas price:           20 gwei
> value sent:          0 ETH
> total cost:          0.01431052 ETH
```

```
   Pausing for 2 confirmations...
   ------------------------------
   > confirmation number: 1 (block: 10184023)
   > confirmation number: 2 (block: 10184024)

   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:          0.05566472 ETH


Summary
=======
> Total deployments:   5
> Final cost:          0.09225052 ETH
```

EtherScan Truffle Verification Plugin

Plugin setup instructions provided by rkalis/truffle-plugin-verify. Thanks to the developer for this useful tool! Really simplifies the process and avoids manually flattening several contract files.

```
npm install --save-dev truffle-plugin-verify
```

Then, within your `truffle-config.js`, make sure you add the following `plugins` section -

```
const HDWalletProvider = require('@truffle/hdwallet-provider');
// WARNING: Do not commit or share the values referencced in this line - use some secret method to store them
const { alchemyApiUrl, mnemonic, etherscanApiKey } = require('./secrets.json'); // Using a secrets.json file
// const mnemonic = fs.readFileSync(".secret").toString().trim(); // Or use a .secret file

module.exports = {
  plugins: [
    'truffle-plugin-verify'
  ],
  api_keys: {
    etherscan: etherscanApiKey
  },
```

Store a `secrets.json` in your project directory to resolve your secrets -

```
{
  "alchemyApiUrl": "https://eth-ropsten.alchemyapi.io/v2/xxxxxxxxxxYOUR_ALCHEMY_API_KEYxxxxxxxxxxx",
  "mnemonic": "word word word word word word word word word word word word",
  "etherscanApiKey": "SOMEREALLYLONGETHERSCANAPIKEY"
}
```

Now, we can simply run `npx truffle run verify <ContractName> --network <NetworkName>` to verify our contracts!

If these steps are not followed carfully, `truffle-plugin-verify` will fail verification due to previous contracts not being verified first. The order of verification is important here, or at least it was for my contract. If I tried to verify in any other order, or without specifying the contract addresses, `Karma` would fail verification.

To verify the above contracts with `truffle-plugin-verify`, we just need to verify them in the order they were deployed. Run the following commands. Notice that when we verify `Karma`, we specify the contract addresses in the order they were deployed. So we verify `Karma@0xb2281089EBf1baB9d701f8697b1eCBaC3319e00F` first, then `Karma@0x957684dC3De2b93154b2561c7bC96875306E39A0` that replaced it in the subsequent deploy. When we jus try to verify `Karma`, we can see that it fails initially, and we can only verify by specifying contract addresses in-order -

```
[kapper@kubuntu-vbox karma]$npx truffle run verify Migrations --network ropsten
Verifying Migrations
Pass - Verified:
https://ropsten.etherscan.io/address/0x8EfAf71d6126b2A0e76A319c92303B3E6Fdc521c#contracts
Successfully verified 1 contract(s).
[kapper@kubuntu-vbox karma]$npx truffle run verify Karma --network ropsten
Verifying Karma
Fail - Unable to verify
Failed to verify 1 contract(s): Karma
[kapper@kubuntu-vbox karma]$npx truffle run verify
Karma@0xb2281089EBf1baB9d701f8697b1eCBaC3319e00F --network ropsten
Verifying Karma@0xb2281089EBf1baB9d701f8697b1eCBaC3319e00F
Pass - Verified:
https://ropsten.etherscan.io/address/0xb2281089EBf1baB9d701f8697b1eCBaC3319e00F#contracts
Successfully verified 1 contract(s).
[kapper@kubuntu-vbox karma]$npx truffle run verify
Karma@0x957684dC3De2b93154b2561c7bC96875306E39A0 --network ropsten
Verifying Karma@0x957684dC3De2b93154b2561c7bC96875306E39A0
Pass - Verified:
https://ropsten.etherscan.io/address/0x957684dC3De2b93154b2561c7bC96875306E39A0#contracts
```

```
Successfully verified 1 contract(s).
```

Notioe that we did not have to verify our `ProxyAdmin` or `TransparentUpgradeableProxy` - these contracts are automatically verified by OpenZeppelin's `truffle-upgrades` plugin.

For quick access, here are links to all the contracts deployed above

Migrations

Karma

Karma (Replacement)

(My) ProxyAdmin

TransparentUpgradeableProxy

And, the resulting token on etherscan is found here - Karma Token

In the next section, we will see how upgrades work for `Karma`. Note that the output shown in the next section was on a different deploy, so the contracts deployed in this section is entirely seperate, and they each have their own `TransparentUpgradeableProxy` and Token on etherscan.

Deploying Token Upgrades

First, we need to define a migration to upgrade to `KarmaV2`. -

> The code for the `KarmaV2` contract is found in the Upgradeable ERC20/Contracts section above.

```
// migrations/4_upgrade_karma.js
const { upgradeProxy } = require('@openzeppelin/truffle-upgrades');


const Karma = artifacts.require('Karma');
const KarmaV2 = artifacts.require('KarmaV2');


module.exports = async function (deployer) {
    const existing = await Karma.deployed();
    await upgradeProxy(existing.address, KarmaV2, { deployer });
};
```

And, to later deploy a `KarmaV3`, we need to add another deploy to our `./migrations/` directory. The file below is an example of what I used to deploy this upgrade.

```
// migrations/5_upgrade_karma.js
const { upgradeProxy } = require('@openzeppelin/truffle-upgrades');


const KarmaV2 = artifacts.require('KarmaV2');
const KarmaV3 = artifacts.require('KarmaV3');


module.exports = async function (deployer) {
   const existing = await KarmaV2.deployed();
   await upgradeProxy(existing.address, KarmaV3, { deployer });
};
```

And, as a further example, here is an even newer version for `Karma` that adds a new function `getAddress` that simply returns the address of the `KarmaV3` contract. This function can then be used to return the address of the current contract that is implementing calls to the `TransparentProxy`, in this and future versions of `Karma`.

```
// contracts/karma-3-ERC20.sol
// Copyright [2021] - [2021], [Shaun Reed] and [Karma] contributors
// SPDX-License-Identifier: MIT


pragma solidity >= 0.8.0;


// ----------------------------------------------------------------------------
// Import ERC Token Standard #20 Interface
//   ETH EIP repo: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md
// ----------------------------------------------------------------------------
import "@openzeppelin/contracts-upgradeable/token/ERC20/ERC20Upgradeable.sol";
import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";


// ----------------------------------------------------------------------------
// Karma Contract
// ----------------------------------------------------------------------------


contract KarmaV3 is Initializable, ERC20Upgradeable
{
   function initialize(string memory name, string memory symbol, uint256 initialSupply) public virtual initializer {
      __ERC20_init(name, symbol);
      _mint(_msgSender(), initialSupply);
```

```
    }

    function isToken() public pure returns (bool)
    {
        return true;
    }

    function getAddress() public view returns (address)
    {
        return address(this);
    }

}
```

Now we are ready to deploy the upgrades to etherscan -

```
[kapper@kubuntu-vbox karma-new]$npx truffle migrate --network ropsten

Compiling your contracts...
===========================
> Compiling ./contracts/Migrations.sol
> Compiling ./contracts/karma-1-ERC20.sol
> Compiling ./contracts/karma-2-ERC20.sol
> Compiling ./contracts/karma-3-ERC20.sol
> Compiling @openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol
> Compiling @openzeppelin/contracts-upgradeable/token/ERC20/ERC20Upgradeable.sol
> Compiling @openzeppelin/contracts-upgradeable/token/ERC20/IERC20Upgradeable.sol
> Compiling @openzeppelin/contracts-upgradeable/token/ERC20/extensions/IERC20MetadataUpgradeable.sol
> Compiling @openzeppelin/contracts-upgradeable/utils/ContextUpgradeable.sol
> Compilation warnings encountered:

    Warning: Function state mutability can be restricted to pure
  --> /home/kapper/Code/karma-new/contracts/karma-2-ERC20.sol:24:5:
   |
24 |     function isToken() public returns (bool)
   |     ^ (Relevant source part starts here and spans across multiple lines).


> Artifacts written to /home/kapper/Code/karma-new/build/contracts
> Compiled successfully using:
   - solc: 0.8.0+commit.c7dfd78e.Emscripten.clang



Starting migrations...
======================
> Network name:    'ropsten'
> Network id:      3
> Block gas limit: 8000000 (0x7a1200)


1_initial_migration.js
======================

   Deploying 'Migrations'
   ----------------------
   > transaction hash:    0x13d88d460d187ef446f48ae52984a8387b06e20ecc3887dbf52a2a958fa7ea29
   > Blocks: 1            Seconds: 12
   > contract address:    0xeB3c4405174931CDDd59E5edfAB03fc46100Ec6D
   > block number:        10179266
   > block timestamp:     1620248502
   > account:             0x9F688c4D72356E230E9db8fA58f6cf981C16E63f
   > balance:             1.74300232575
   > gas used:            245600 (0x3bf60)
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.004912 ETH

   Pausing for 2 confirmations...
   ------------------------------
   > confirmation number: 1 (block: 10179267)
   > confirmation number: 2 (block: 10179268)

   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:            0.004912 ETH


2_deploy_karma.js
```

```
==================

   Deploying 'Karma'
   -----------------
   > transaction hash:    0xa21e6335e217cf4056541f842957e8370009c19889f8c1455dc0de4b11407ea3
   > Blocks: 0            Seconds: 24
   > contract address:    0xB2425Ea8087233A8cD4140E1480F60EB57C133aE
   > block number:        10179270
   > block timestamp:     1620248640
   > account:             0x9F688c4D72356E230E9db8fA58f6cf981C16E63f
   > balance:             1.71041050575
   > gas used:            1583678 (0x182a3e)
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.03167356 ETH

   Pausing for 2 confirmations...
   ------------------------------
   > confirmation number: 1 (block: 10179271)
   > confirmation number: 2 (block: 10179272)

   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:          0.03167356 ETH


3_deploy_proxy.js
==================

   Deploying 'TransparentUpgradeableProxy'
   ---------------------------------------
   > transaction hash:    0x774ddb93ed3b219202195821e43939f031320d441cf78e8bbf497724b06f2b05
   > Blocks: 1            Seconds: 52
   > contract address:    0xdCDA9d33Eb6eEf5C748743Bb1e2B7FBFBc500904
   > block number:        10179275
   > block timestamp:     1620248756
   > account:             0x9F688c4D72356E230E9db8fA58f6cf981C16E63f
   > balance:             1.69552372575
   > gas used:            715526 (0xaeb06)
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.01431052 ETH

   Pausing for 2 confirmations...
   ------------------------------
   > confirmation number: 1 (block: 10179277)
   > confirmation number: 2 (block: 10179278)

   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:          0.01431052 ETH



4_upgrade_karma.js
==================

   > Saving migration to chain.
   -------------------------------------
   > Total cost:                   0 ETH



   5_upgrade_karma.js
==================

   Deploying 'KarmaV3'
   -------------------
   > transaction hash:    0x3c446b4ac1e647d3145b5e187e03d181f5de101fe568271e6ded27c90fe1ca0e
   > Blocks: 0            Seconds: 12
   > contract address:    0x9e3Be3194de7A033f82e7aC121b1036Dd817f4c7
   > block number:        10179297
   > block timestamp:     1620248992
   > account:             0x9F688c4D72356E230E9db8fA58f6cf981C16E63f
   > balance:             1.62875240575
   > gas used:            1621063 (0x18bc47)
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.03242126 ETH
   Pausing for 2 confirmations...
   ------------------------------
   > confirmation number: 1 (block: 10179298)
   > confirmation number: 2 (block: 10179299)

   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:          0.03242126 ETH


Summary
=======
> Total deployments:   1
> Final cost:          0.08331734 ETH
```

> In the output above, you'll notice the `4_upgrade_karma.js` deploy did not provide a contract address or create a transaction. This is because I already had this version deployed on the ropsten testnet when writing this page. As an example, I added the third version to the deploy to show how to upgrade subsequent contracts.

For quick access, here's links to each contract deployed above on EtherScan:

(My) ProxyAdmin

TransparentUpgradeableProxy

Karma (V1)

KarmaV2

KarmaV3

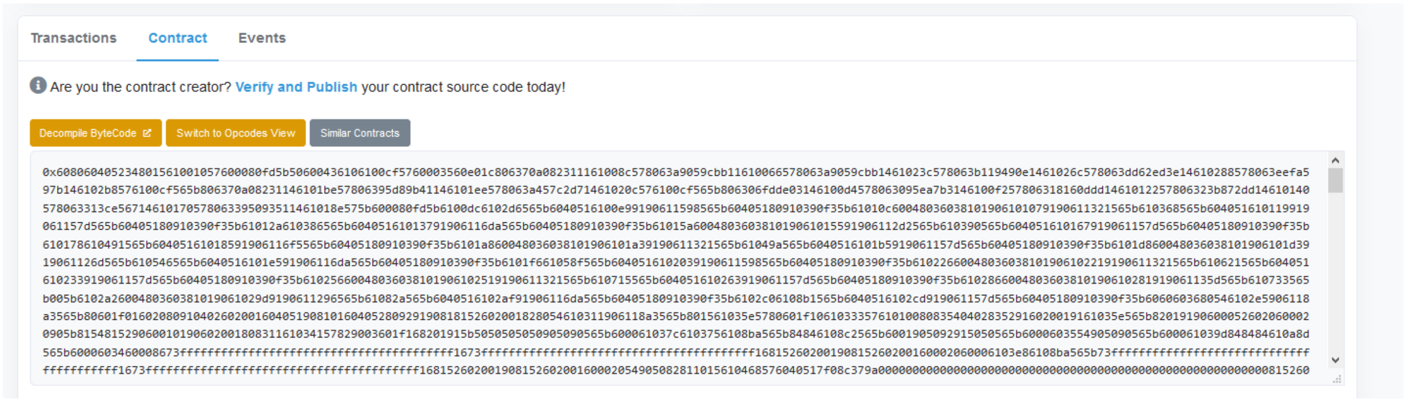
And, the resulting token on etherscan is found here - Karma Token

Manual EtherScan Verification

If you are unable to get `truffle-plugin-verify` to work for your contracts, you can alternatively flatten them manually and submit them to etherscan yourself for verification. This section will cover how to do this.

Verification of contracts on etherscan is not so straight-forward the first time around, or at least it wasn't for me. Head over to etherscan, make sure you are viewing the correct network, and enter the deployed contract address for `Karma` (Or, in your case, enter the contract address that replaced `Karma`)

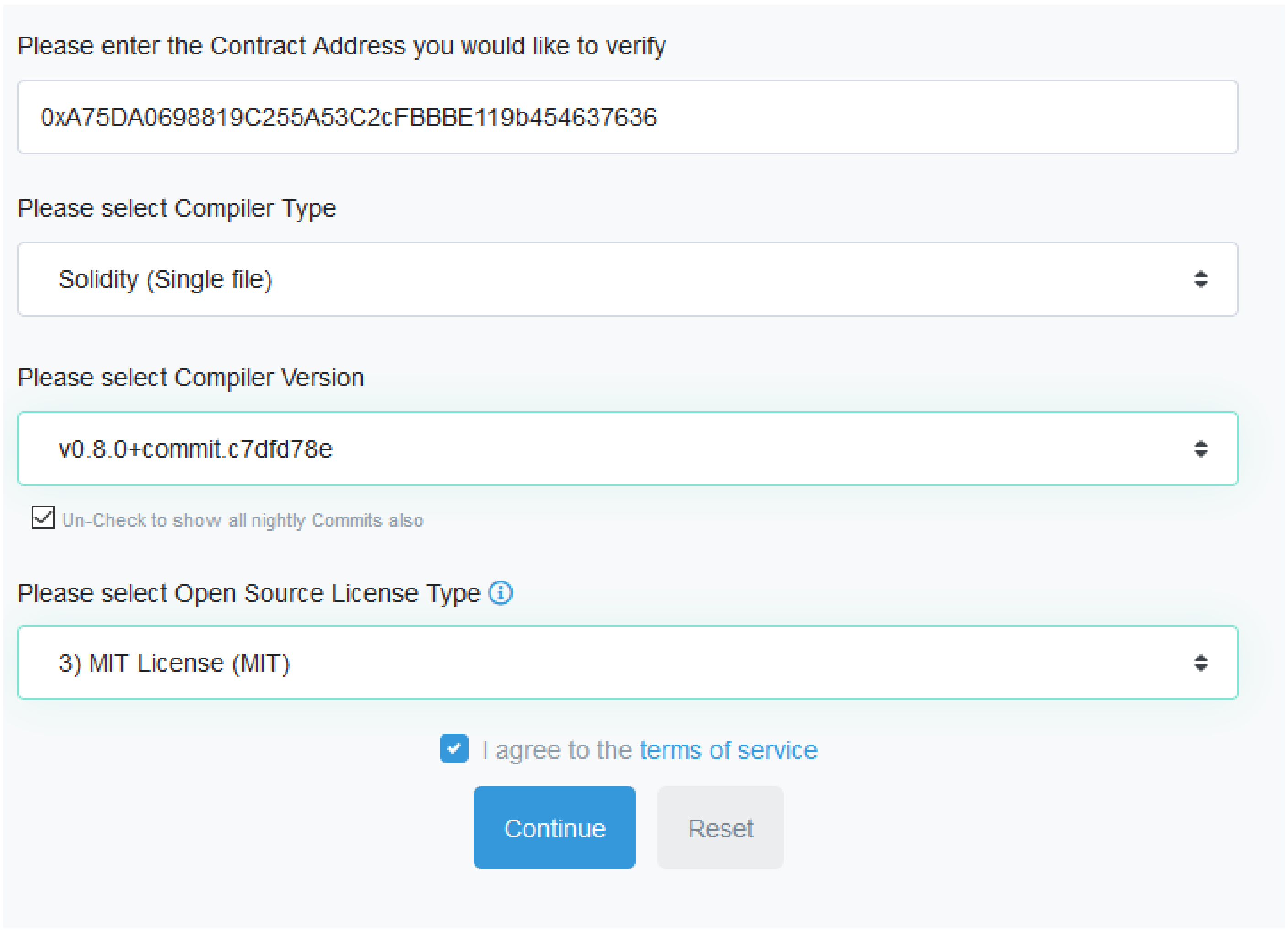
Here is the contract page for Karma that we deployed on ropsten in the output above. Notice the verification status of `Karma` is Verified. Compare this page to the same page on your contract. The `Karma` contract should allow you to call functions within the `read contract` and `write contract` pages, while an unverified contract will not provide access to these functions. So, we need to verify our contracts on etherscan. When we try to view an unverified contract, we will be presented with a link to verify that looks similar to the page below.



Are you the contract creator? Verify and Publish your contract source code today!

```
0x608060405234801561001057600080fd5b50600436106100cf5760003560e01c806370a082311161008c578063a9059cbb11610066578063a9059cbb1461023c578063b119490e1461026c578063dd62ed3e14610288578063eefa5
97b146102b8576100cf565b806370a08231146101be57806395d89b41146101ee578063a457c2d71461020c576100cf565b806306fdde03146100d4578063095ea7b3146100f257806318160ddd1461012257806323b872dd14610140
578063313ce56714610170578063395093511461018e575b600080fd5b6100dc6102d6565b6040516100e99190611598565b60405180910390f35b61010c60048036038101906101079190611321565b610368565b604051610119919
061157d565b60405180910390f35b61012a610386565b60405161013791906116da565b60405180910390f35b61015a600480360381019061015591906112d2565b610390565b604051610167919061157d565b60405180910390f35b
610178610491565b60405161018591906116f5565b60405180910390f35b6101a860048036038101906101a39190611321565b61049a565b6040516101b5919061157d565b60405180910390f35b6101d860048036038101906101d39
19061126d565b610546565b6040516101e591906116da565b60405180910390f35b6101f661058f565b6040516102039190611598565b60405180910390f35b6102266004803603810190610221919061 1321565b610621565b604051
610233919061157d565b60405180910390f35b61025660048036038101906102519190611321565b610715565b604051610263919061157d565b60405180910390f35b6102866004803603810190610281919061135d565b610733565
b005b6102a2600480360381019061029d9190611296565b61082a565b6040516102af91906116da565b60405180910390f35b6102c06108b1565b6040516102cd919061157d565b60405180910390f35b6060603680546102e5906118
a3565b80601f0160208091040260200160405190810160405280929190818152602001828054610311906118a3565b801561035e5780601f106103335761010080835404028352916020019161035e565b82019190600052602060002
0905b815481529060010190602001808311610341578290036 01f168201915b5050505050905090565b600061037c6103756108ba565b84846108c2565b6001905092915050565b6000603554905090565b600061039d848484610a8d
565b6000603460008673ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260200160002060006103e86108ba565b73fffffffffffffffffffffffffffff
ffffffffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260200160002054905082811015610468576040517f08c379a000000000000000000000000000000000000000000000000000000000815260
```

Click `Verify and Publish`, and you will see a page that asks for several options you configured when setting up your project and writing your contracts. Select the options that match your token. My options for `Karma` are seen in the screenshot below. Please disregard the contract address in this image, as it does not reflect any address we have linked to on this page. The compiler type, version, and license are all still relevant to `Karma`.



Please enter the Contract Address you would like to verify

0xA75DA0698819C255A53C2cFBBBE119b454637636

Please select Compiler Type

Solidity (Single file)

Please select Compiler Version

v0.8.0+commit.c7dfd78e

✓ Un-Check to show all nightly Commits also

Please select Open Source License Type

3) MIT License (MIT)

✓ I agree to the terms of service

Now, we will be presented with a screen that asks up to input the source code for the contract. **Including other contracts with an `import` is not allowed here**, and to verify simply the easiest and most effective way is to 'flatten' your `.sol` contract manually (and carefully).

At each `import` statement, recursively, simply copy-paste the code from the import in-place of the import statement itself. Remove any `SPDX-License-Identifier` beyond the initial `SPDX-License-Identifier` you declare (or should declare) at the very top of your own contract. Be sure not to leave *any* imports in the flattened contract, or verification will fail. If your contracts use arguments in the constructor, you will have to research ABI format for inputting your constructor arguments. `Karma` does not use any arguments in the constructor, so there is no need for me to do this in this example.

Here is a pastebin example of my input that verified KarmaV1

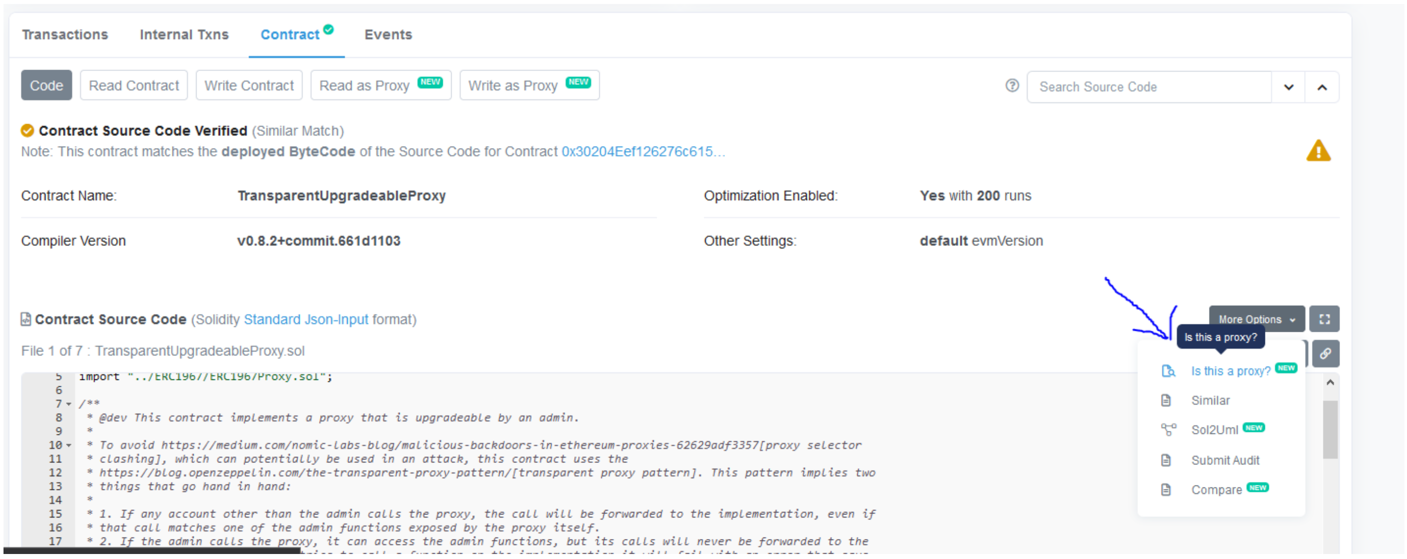
Here is a pastebin example of my input that verified KarmaV2

Here is a pastebin example of my input that verified KarmaV3

Once the verification of these implementation contracts is complete, we are ready for the final step - verifying the `TransparentUpgradeableProxy`. To do this, navigate to the relative contract address on etherscan for your deploy. For my example, the contract for the TransparentUpgradeableProxy can be found at this link

This part is very easy to miss on etherscan, and wraps up all the work we've done so far

In the image below, click `More Options` and then click `Is this a proxy?` in the context menu that pops up.



Contract Source Code Verified (Similar Match)
Note: This contract matches the **deployed ByteCode** of the Source Code for Contract 0x30204Eef126276c615...

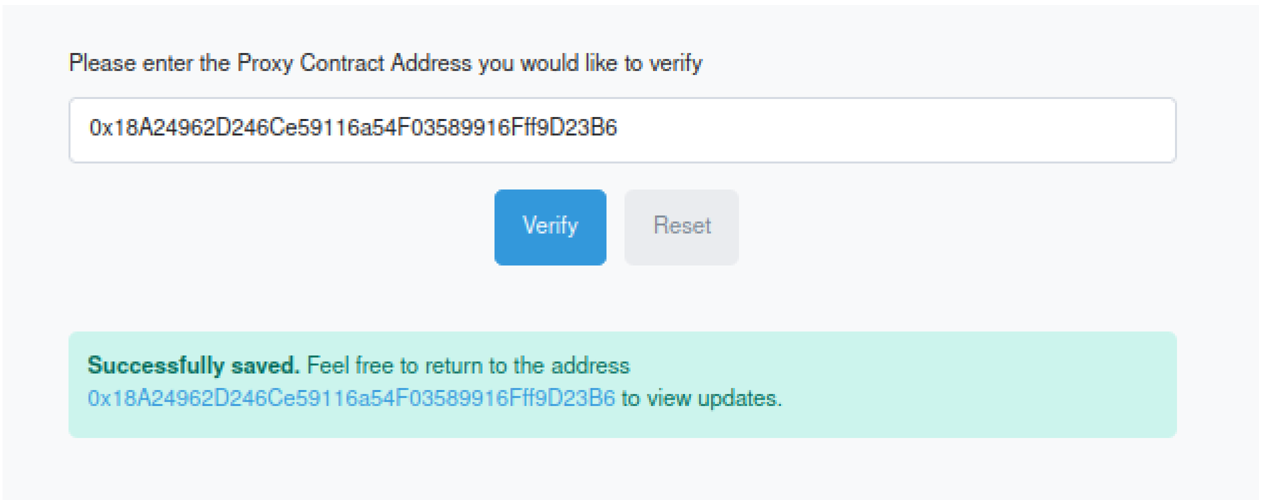
Contract Name:	**TransparentUpgradeableProxy**	Optimization Enabled:	**Yes** with **200** runs
Compiler Version	**v0.8.2+commit.661d1103**	Other Settings:	**default** evmVersion

Contract Source Code (Solidity Standard Json-Input format)

File 1 of 7 : TransparentUpgradeableProxy.sol

```
import "../ERC1967/ERC1967Proxy.sol";

/**
 * @dev This contract implements a proxy that is upgradeable by an admin.
 *
 * To avoid https://medium.com/nomic-labs-blog/malicious-backdoors-in-ethereum-proxies-62629adf3357[proxy selector
 * clashing], which can potentially be used in an attack, this contract uses the
 * https://blog.openzeppelin.com/the-transparent-proxy-pattern/[transparent proxy pattern]. This pattern implies two
 * things that go hand in hand:
 *
 * 1. If any account other than the admin calls the proxy, the call will be forwarded to the implementation, even if
 * that call matches one of the admin functions exposed by the proxy itself.
 * 2. If the admin calls the proxy, it can access the admin functions, but its calls will never be forwarded to the
```

This will lead you to the following verification page, which automatically fills out the contract address of the `TransparentUpgradeableProxy` and allows you to submit it for verification as a proxy contract. Click `Verify`, and if all has been done correctly, the `TransparentUpgradeableProxy` will verify with the message below!

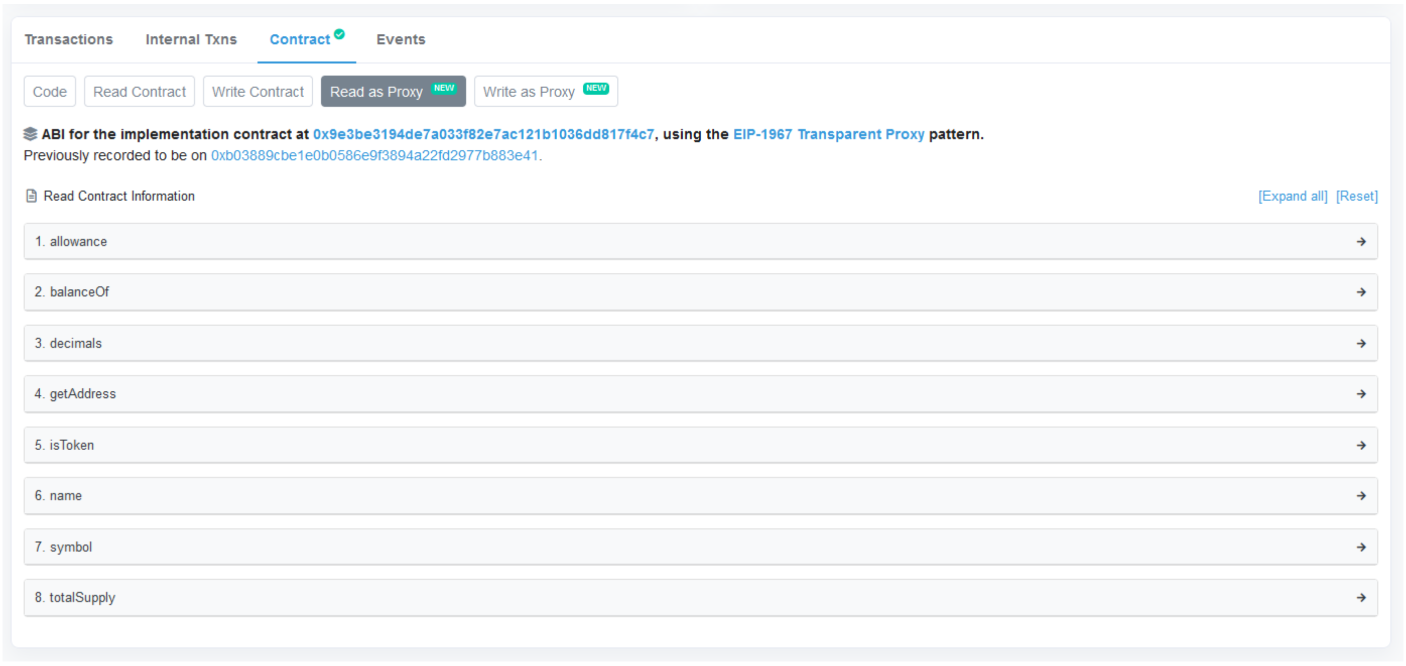


Please enter the Proxy Contract Address you would like to verify

0x18A24962D246Ce59116a54F03589916Fff9D23B6

Successfully saved. Feel free to return to the address 0x18A24962D246Ce59116a54F03589916Fff9D23B6 to view updates.

That's it! We have deployed an upgradeable ERC20 token on the Ethereum ropsten testnet. After verifying your `TransparentUpgradeableProxy`, you should notice the addition of the `Read as Proxy` and `Write as Proxy` options when viewing the `TransparentUpgradeableProxy` on etherscan, as seen in the image below.



ABI for the implementation contract at 0x9e3be3194de7a033f82e7ac121b1036dd817f4c7, using the EIP-1967 Transparent Proxy pattern.
Previously recorded to be on 0xb03889cbe1e0b0586e9f3894a22fd2977b883e41.

Read Contract Information [Expand all] [Reset]

1. allowance
2. balanceOf
3. decimals
4. getAddress
5. isToken
6. name
7. symbol
8. totalSupply

We should notice that the implementation contract referred to in bold is the address or the most recent `Karma`, which is `KarmaV3`. We can be sure of this by both the contract address and the addition of the `isToken` and `getAddress` functions seen under `Read as Proxy`. Pretty cool!

We now have a single contract that can refer to the most recent implementation we have defined on the ropsten testnet. Hopefully, some of this information has helped you to deploy your own token.