Supplying Assets to the Compound Protocol

Quick Start Guide

Adam Bavosa Follow
Feb 12, 2020 · 14 min read

The Compound Protocol is a series of interest rate markets running on the Ethereum blockchain. When users and applications supply an asset to the Compound Protocol, they begin earning a variable interest rate instantly. Interest accrues every Ethereum block (currently ~13 seconds), and users can withdraw their principal plus interest anytime.

Under the hood, users are contributing their assets to a large pool of liquidity (a "market") that is available for other users to borrow, and they share in the interest that borrowers pay back to the pool.

When users supply assets, they receive cTokens from Compound in exchange. cTokens are ERC20 tokens that can be redeemed for their underlying assets at any time. As

interest accrues to the assets supplied, cTokens are redeemable at an exchange rate (relative to the underlying asset) that constantly increases over time, based on the rate of interest earned by the underlying asset.

Non-technical users can interact with the Compound Protocol using an interface like Argent, Coinbase Wallet, or app.compound.finance; developers can create their own applications that interact with Compound's smart contracts.

In this guide, we're going to walk through **supplying assets via Web3.js JSON RPC and via proxy smart contracts that live on the blockchain**. These are two methods in which developers can write software to utilize the Compound Protocol.

There are examples in **JavaScript** and also **Solidity**.

Table of Contents for This Guide

- Compound Markets
- Connecting to the Ethereum Network
- Supplying on a Localhost Network
- Supplying on a Public Network
- How to Supply ETH to the protocol via Web3.js
- How to Supply a Supported ERC20 Token to the protocol via Solidity

If you are new to Ethereum, we suggest that you start by Setting up your Development Environment for Ethereum.

> *All of the **code** referenced in this guide can be found in this **GitHub Repository**: Quick Start: Supplying Assets to the Compound Protocol.*

To copy the repository to your computer, run this on the command line after you've installed git:

```
git clone git@github.com:compound-developers/compound-supply-examples.git
```

Compound Markets

The Compound Protocol enables developers to build innovative products on DeFi. So

far, we've seen crypto wallets equipped with savings APRs, a no-loss lottery system, an interest-earning system for donation income, and more.

The smart contracts that power the protocol are deployed to the Ethereum blockchain. This means that at the time of this guide's writing, the only types of assets that Compound can support are Ether and ERC-20 tokens.

The currently supported assets are listed here https://compound.finance/markets. Based on the different implementation of Ether (ETH) and ERC-20 tokens, we have to utilize two similar processes:

- The ETH supply method
- The ERC20 token supply method

Like mentioned earlier, when someone supplies an asset to the protocol, they are given **cTokens** in exchange. The method for getting **cETH** is different from the method for getting **cDAI**, **cUNI**, or any other cToken for an ERC-20 asset. We'll run through code examples and explanations for the two different asset supply methods.

When supplying Ether to the Compound protocol, an application can send ETH directly to the payable **mint** function in the cEther contract. Following that mint, cEther is minted for the wallet or contract that invoked the mint function. Remember that if you are calling this function from another smart contract, that contract needs a **payable** function in order to receive ETH when you redeem the cTokens later.

The operation is slightly different for cERC20 tokens. In order to mint cERC20 tokens, the invoking wallet or contract needs to first call the **approve** function on the **underlying token's contract**. All ERC20 token contracts have an **approve** function.

The approval needs to indicate that the corresponding cToken contract is permitted to take *up to the specified amount* from the sender address. Subsequently, when the **mint** function is invoked, the cToken contract retrieves the indicated amount of underlying tokens from the sender address, based on the prior approve call.

Example code for each method (JS and Solidity) is available, open source, in the GitHub Repository linked above.

Connecting to the Ethereum Network

You will need to use the contract address for the particular network that you're

developing on; start by identifying the contract address for each network in the Docs. In this guide, we'll create a fork of Mainnet, which will run on our localhost; copy the Mainnet addresses.

If you want to use a public test net (like Ropsten, Göerli, Kovan, or Rinkeby), make an Infura account at https://infura.io/ to get your API key. If you are using your own localhost test net, or the production mainnet, we will also use Infura.

If you are not hosting your own Ethereum node to access the blockchain, make an Infura account before continuing.

For more on connecting to a public Ethereum network, see the instructions in Setting up your Development Environment for Ethereum.

Supplying to the Compound Protocol on a Localhost Network

To run an Ethereum local test net on your machine, we will fork the Main network (a.k.a Homestead or Mainnet). This means that you can interact with the production smart contracts in a test environment. **No real ETH will be used and no modifications to the production blockchain will occur**. If you haven't already, install Node.js. Click here to install the LTS of Node.js and NPM.

Let's install and initialize **Ganache CLI**.

```
npm i -g ganache-cli

## or for yarn fans: yarn global add ganache-cli
```

Run this command in **a second command line window** before you start running the code referenced later in this guide. The command spins up a test Ethereum blockchain on your localhost. **Be sure to supplant your Infura project ID into the JSON RPC Provider URL.**

```
ganache-cli \
  -f https://mainnet.infura.io/v3/<YOUR INFURA API KEY HERE> \
  -m "clutch captain shoe salt awake harvest setup primary inmate
ugly among become" \
  -i 1 \
  -u 0x9759A6Ac90977b93B58547b4A71c78317f391A28
```

A quick explanation of each of the command line flags:

- `-f` **Forks** the Main Ethereum network to your local machine for development and testing.
- `-m` Runs Ganache with an Ethereum key set based on the **mnemonic** passed. The first 10 addresses have 100 test ETH in their balance on the local test net every time you boot Ganache. **Do not use this mnemonic anywhere other than your localhost test net.**
- `-i` Sets an explicit network ID to avoid confusion and errors.
- `-u` Unlocks an address so you can write to your localhost test blockchain without knowing that address's private key. We are unlocking the above address **so we can mint our own test DAI** on our localhost test net (more on this later).

Once you have run Ganache CLI on your command line, it will log 10 wallet addresses, and 10 private keys. Each of the wallets will have 100 test ETH in them which can be used for executing smart contracts locally. Copy and save the first private key.

Supplying to Compound on a Public Network

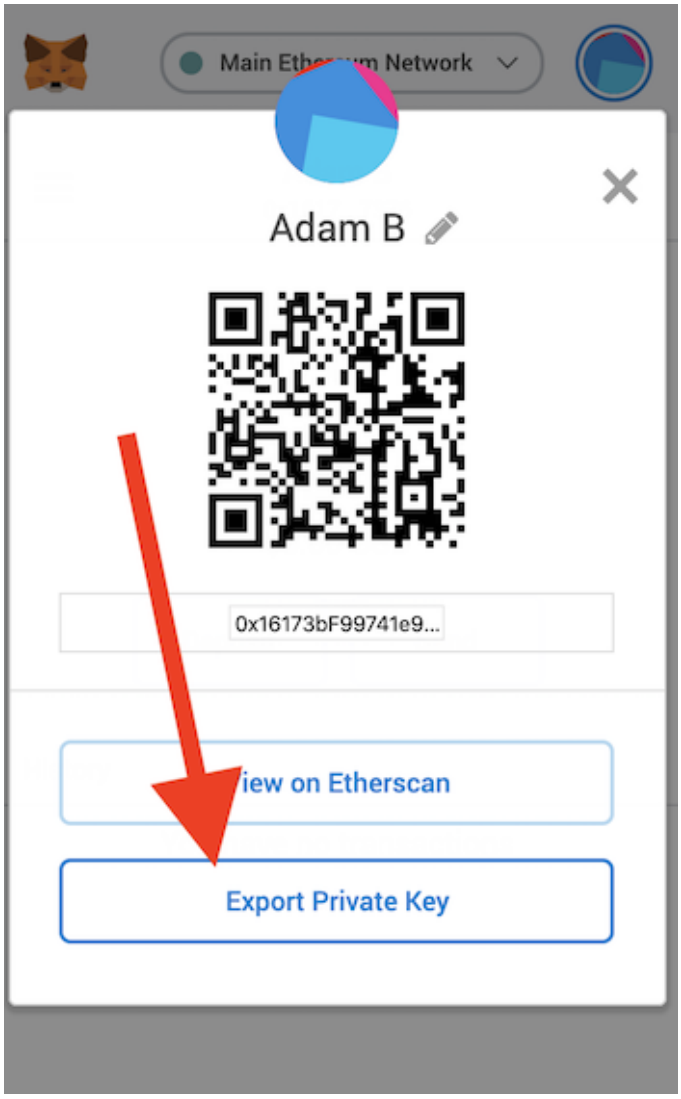
If you are supplying to the protocol on the Mainnet, Ropsten, Göerli, Kovan, or Rinkeby, you should have already located and copied the **Compound contract address** for that network (see how above). You'll need it for later.

You also should have collected some ETH for that network by purchasing/mining (Main), or a test net's faucet (all the others). This is not necessary when using a localhost fork.

For example, here is Ropsten's faucet https://faucet.ropsten.be/. You can send yourself 1 ETH every 24 hours from a single IP address. This is test ETH that is only applicable to the Ropsten test network.

Next, copy and safely store your wallet's private key. **Don't do this if you are only testing on your localhost.** The private key is used to sign transactions that are sent on the Ethereum network. The purpose of this is to certify that the transaction was created and submitted by a unique wallet.

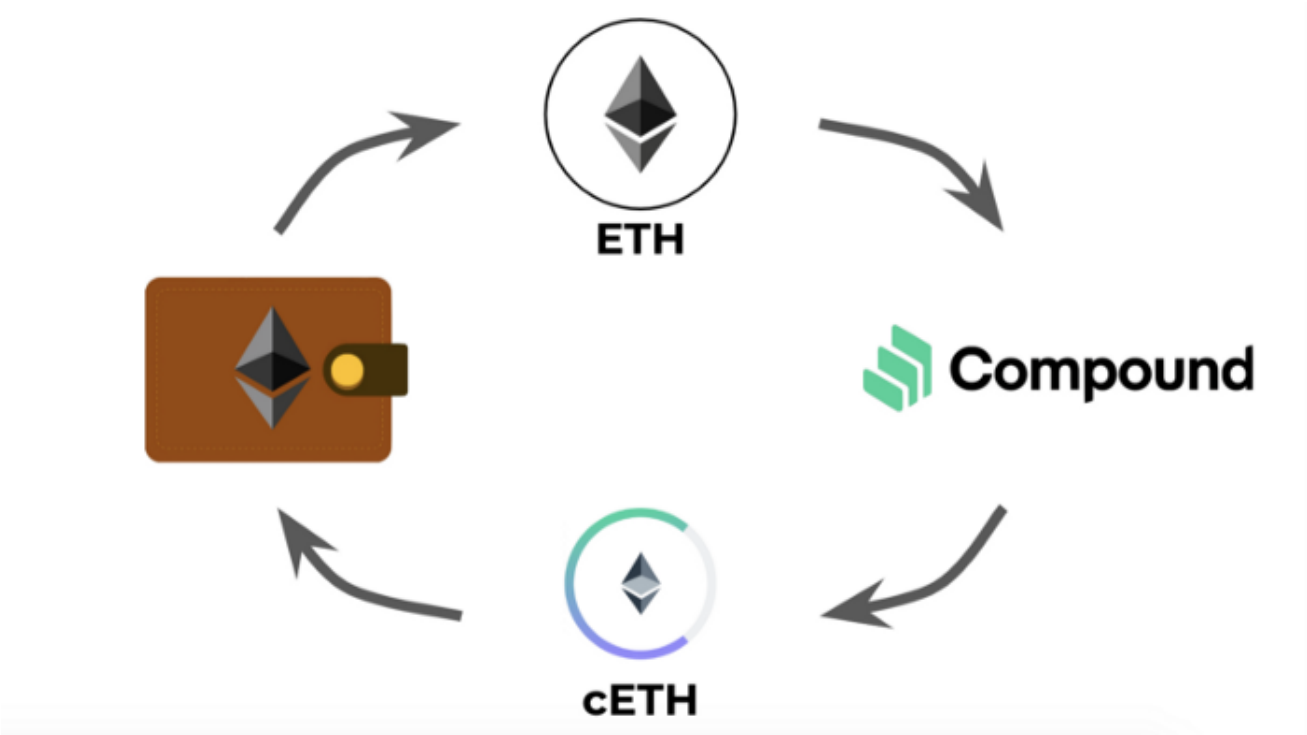
If you are using MetaMask for your Ethereum wallet, open the menu, click the 3 dots on the right, Account Details, Export Private Key, and input your MetaMask password. This will reveal your private key. **Keep it safe!** Copy this value and save it for later.



It is a **best practice** to store a key like this as an environment variable on your local machine. When a key is stored as an environment variable, it can be referenced in code files by a variable name, instead of explicitly with a string. This promotes code cleanliness, and **reduces the risk of exposing your secret**.

Again, if you are only testing smart contracts on your localhost Ganache today, **don't get your MetaMask private key**. We'll rely on the private key from the Ganache command line log.

How to Supply ETH to Compound via Web3.js



Supplying Ether (ETH) to the Compound Protocol is as easy as calling the "mint" function in the Compound cEther smart contract. The "mint" function transfers ETH to the Compound contract address, and mints cETH tokens. The cETH tokes are transferred to the wallet of the supplier.

Remember that the amount of ETH that can be exchanged for cETH increases every Ethereum block, which is about every 13 seconds. **There is no minimum or maximum amount of time that suppliers need to keep their asset in the protocol**. See the varying exchange rate for each cToken by clicking on one at https://compound.finance/markets.

For more information on cToken concepts see the cToken documentation.

In order to call the mint function, you need to first:

- Have ETH in your Ethereum wallet.
- Find your Ethereum wallet's private key.
- Connect to the network via Infura API key (see above section **Connecting to the Ethereum Network**)

There are several programming languages that have Ethereum Web3 libraries, but the most popular at the time of this guide's writing is **JavaScript**.

We'll be using Node.js JavaScript to call the mint function. The following code snippets are from this Node.js file in the supplying assets guide GitHub Repository. **Web browser JavaScript** is nearly identical to these code examples.

Let's import Web3.js, and initialize the Web3 object. It's pointing to our localhost's Ganache, which has 100 test ETH in each of the test wallets. We get the same 10 test wallet addresses every time we run Ganache CLI with the mnemonic in the above command (from the **Connecting to the Ethereum Network** section).

If you are using a public network (Ropsten, Kovan, etc.), make sure your wallet has ETH, and that you have your wallet private key stored as an environment variable. Also, have your Infura API key ready if you are deploying to a public test net.

Replace the HTTP provider URL in the Web3 declaration line with the appropriate network's provider if you are not using the Ganache test environment.

```
1  const Web3 = require('web3');
2  const web3 = new Web3('http://127.0.0.1:8545');
```

supply-eth-via-web3.js hosted with ❤ by **GitHub** **view raw**

Next, we'll add our wallet's private key as a variable. It's a best practice to access this as an environment variable.

```
1  // Your Ethereum wallet private key
2  const privateKey = process.env.myWalletPrivateKey;
3  // Add your Ethereum wallet to the Web3 object
4  web3.eth.accounts.wallet.add('0x' + privateKey);
5  const myWalletAddress = web3.eth.accounts.wallet[0].address;
```

supply-eth-via-web3.js hosted with ❤ by **GitHub** **view raw**

If you are writing **web browser JavaScript instead of Node.js**, you can add the user's private key to the Web3 object by using the **ethereum.enable()** command. Here is the alternative code snippet.

```
1  // Add your Ethereum wallet to the Web3 object
2  ethereum.enable();
3  const myWalletAddress = web3.eth.accounts.wallet[0].address;
```

supply-eth-via-web3.js hosted with ❤ by **GitHub** **view raw**

Next we'll make some variables for the contract address and the contract ABI. The contract addresses are posted on this page: https://compound.finance/docs#networks. Remember to use the mainnet address if you are testing with Ganache CLI. The ABI is the same regardless of the Ethereum network that we are using.

```
// Main Net Contract for cETH (the supply process is different for cERC20 tokens)
const contractAddress = '0x4ddc2d193948926d02f9b1fe9e1daa0718270ed5';
const abiJson = [{"constant":true,"inputs":[],"name":"name","outputs":[{"name":"","ty
const compoundCEthContract = new web3.eth.Contract(abiJson, contractAddress);
```

supply-eth-via-web3.js hosted with ❤ by **GitHub** **view raw**

The next section of code is where the magic happens. The first call in the main function supplies our ETH to the protocol by calling the **mint** function, which mints cETH. The cETH is transferred to our wallet address.

```
const ethDecimals = 18; // Ethereum has 18 decimal places

const main = async function() {
  let ethBalance = await web3.eth.getBalance(myWalletAddress) / Math.pow(10, ethDeci
  console.log("My wallet's ETH balance:", ethBalance, '\n');

  console.log('Supplying ETH to the Compound Protocol...', '\n');
  // Mint some cETH by supplying ETH to the Compound Protocol
  await cEthContract.methods.mint().send({
    from: myWalletAddress,
    gasLimit: web3.utils.toHex(150000),
    gasPrice: web3.utils.toHex(20000000000), // use ethgasstation.info (mainnet only
    value: web3.utils.toHex(web3.utils.toWei('1', 'ether'))
  });

  console.log('cETH "Mint" operation successful.', '\n');

```

supply-eth-via-web3.js hosted with ❤ by **GitHub** **view raw**

The three subsequent function calls are not necessary, but they are here for illustration. The first method calls a getter function in the Compound contract that shows how much **underlying ETH** our cToken balance entitles us to. The second function shows our wallet's **cToken balance**. The third function gets the current **exchange rate** of cETH to ETH.

```
const balanceOfUnderlying = web3.utils.toBN(await cEthContract.methods
    .balanceOfUnderlying(myWalletAddress).call()) / Math.pow(10, ethDecimals);

  console.log("ETH supplied to the Compound Protocol:", balanceOfUnderlying, '\n');

  let cTokenBalance = await cEthContract.methods.balanceOf(myWalletAddress).call() /

  console.log("My wallet's cETH Token Balance:", cTokenBalance, '\n');

  let exchangeRateCurrent = await cEthContract.methods.exchangeRateCurrent().call();
  exchangeRateCurrent = exchangeRateCurrent / Math.pow(10, 18 + ethDecimals - 8);
  console.log("Current exchange rate from cETH to ETH:", exchangeRateCurrent, '\n');

```

supply-eth-via-web3.js hosted with ❤ by **GitHub** **view raw**

Our code sends 1 ETH to the contract, and gives our wallet cETH. The ratio of cETH to ETH should be in the ballpark of 50 to 1. Remember that the exchange rate of underlying to cToken **increases** over time.

Lastly, after the supply operation is complete, we'll redeem our cTokens. This is what a user or application will do when they want to withdraw their crypto asset from the Compound protocol.

The first method, **redeem**, redeems based on the cToken amount passed to the function call.

The second method, **redeemUnderlying**, which is commented out, redeems ETH based on the amount passed to the function call.

```
    console.log('Redeeming the cETH for ETH...', '\n');

    console.log('Exchanging all cETH based on cToken amount...', '\n');
    await cEthContract.methods.redeem(cTokenBalance * 1e8).send({
      from: myWalletAddress,
      gasLimit: web3.utils.toHex(500000),
      gasPrice: web3.utils.toHex(20000000000), // use ethgasstation.info (mainnet only
    });

    // console.log('Exchanging all cETH based on underlying ETH amount...', '\n');
    // let ethAmount = web3.utils.toWei(balanceOfUnderlying).toString()
    // await cEthContract.methods.redeemUnderlying(ethAmount).send({
    //   from: myWalletAddress,
    //   gasLimit: web3.utils.toHex(150000),
    //   gasPrice: web3.utils.toHex(20000000000), // use ethgasstation.info (mainnet o
    // });

    cTokenBalance = await cEthContract.methods.balanceOf(myWalletAddress).call() / 1e8
    console.log("My wallet's cETH Token Balance:", cTokenBalance);

    ethBalance = await web3.eth.getBalance(myWalletAddress) / Math.pow(10, ethDecimals
    console.log("My wallet's ETH balance:", ethBalance, '\n');
  }
```

supply-eth-via-web3.js hosted with ❤ by **GitHub** **view raw**

Finally, we execute the main function and declare an error handler.

```
main().catch((err) => {
  console.error(err);
});
```

supply-erc20-via-solidity.js hosted with ❤ by **GitHub** **view raw**

If you cloned the GitHub repository, be sure to run **npm install** in the root directory of the project **before** you try to run the script.

Here is the command for running the script from the root directory of the project:

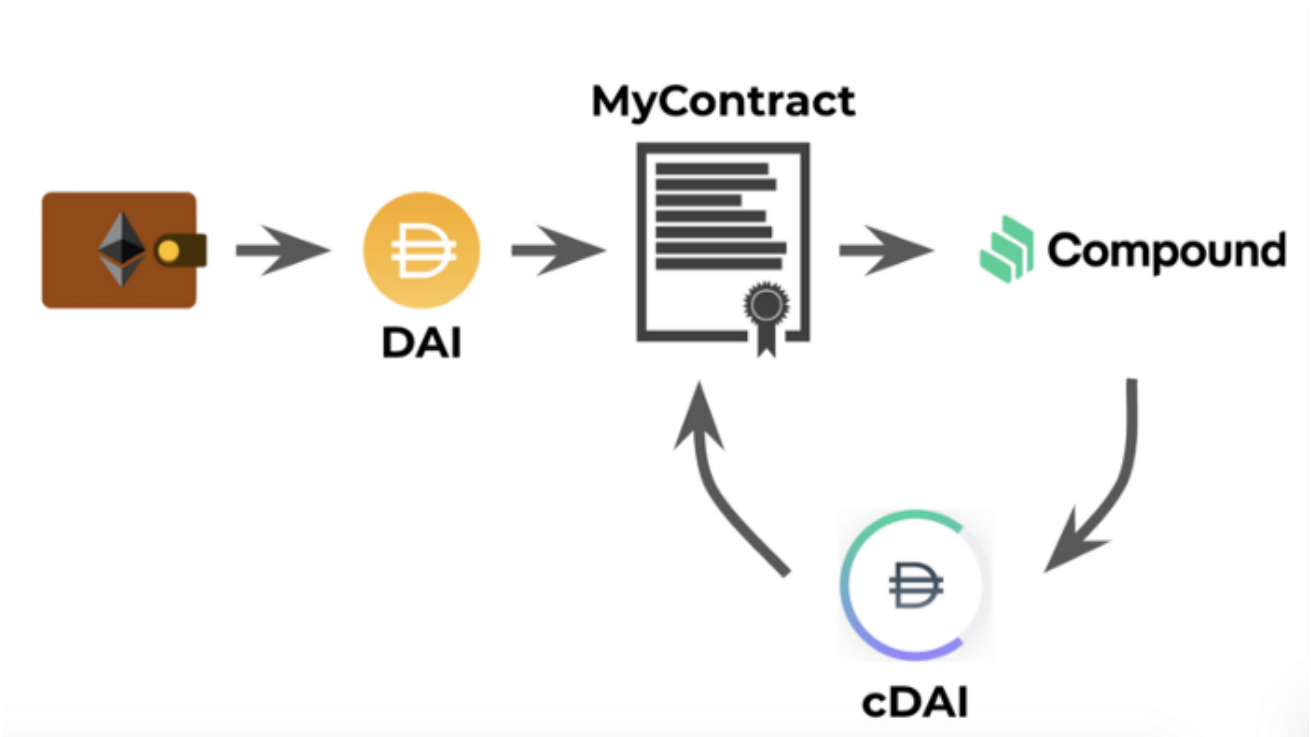
```
node web3-js-examples/supply-eth-via-web3.js
```

Script example output:

```
$ node web3-js-examples/supply-eth-via-web3.js
My wallet's ETH balance: 99.97423756

Supplying ETH to the Compound Protocol...

cETH "Mint" operation successful.

ETH supplied to the Compound Protocol: 0.9999999998672074

My wallet's cETH Token Balance: 49.93244926

Current exchange rate from cETH to ETH: 0.020027056847545625

Redeeming the cETH for ETH...

Exchanging all cETH based on cToken amount...

My wallet's cETH Token Balance: 0
My wallet's ETH balance: 99.96892080053026
```

supply-eth-via-web3-output.txt hosted with ❤ by GitHub view raw

How to Supply a Supported ERC20 Token to Compound via Solidity



The following will run through an example of adding an ERC20 token to the Compound protocol using Solidity smart contracts. The full Solidity file can be found in the project

GitHub repository.

Here's an overview of supplying a token to the Compound Protocol with Solidity:

Prerequisites

- Get some ETH into your own Ethereum wallet by purchasing/mining (or faucets on test nets). This will be used for gas costs. If you're using Ganache CLI on localhost, you're ready.
- Get some ERC20 token, in this case Dai. If you are working in the production environment, purchase some Dai for your Ethereum wallet. If you are working with a Ganache CLI test blockchain, follow the instructions in the section **Minting Test Net Dai** to get your test wallet some Dai.
- Get the address of the ERC20 contract.
- Get the address of the Compound cToken contract. See **Dai** on this page: https://compound.finance/docs#networks.

Order of Operations

- You **transfer** Dai from your wallet to your custom contract. This is not done in Solidity, but instead with Web3.js and JSON RPC.
- You call your custom contract's function for supplying to the Compound Protocol.
- Your custom contract's function calls the approve function from the original ERC20 token contract. This allows an amount of the token to be withdrawn by cToken from your custom contract's token balance.
- Your custom contract's function calls the **mint** function in the Compound cToken contract.
- Finally, we call your custom contract's function for redeeming, to get the ERC20 token back.

Let's get started. First we'll walk through the code in our Solidity file, **MyContracts.sol**.

```
pragma solidity ^0.5.12;


interface Erc20 {
    function approve(address, uint256) external returns (bool);

    function transfer(address, uint256) external returns (bool);
}


interface CErc20 {
    function mint(uint256) external returns (uint256);

    function exchangeRateCurrent() external returns (uint256);

    function supplyRatePerBlock() external returns (uint256);

    function redeem(uint) external returns (uint);

    function redeemUnderlying(uint) external returns (uint);
}
```

MyContracts.sol hosted with ❤ by **GitHub** **view raw**

We added contract interfaces. The first is for our ERC20 token contract, and the second is for Compound's corresponding cToken contract.

We'll be able to call the production versions of the 3rd party contracts using these definitions. We need to initialize them with the production address of the deployed contracts, which we pass to each of the functions in **MyContract**.

```
contract MyContract {
    event MyLog(string, uint256);

    // omitting some code here...

    function supplyErc20ToCompound(
        address _erc20Contract,
        address _cErc20Contract,
        uint256 _numTokensToSupply
    ) public returns (uint) {
        // Create a reference to the underlying asset contract, like DAI.
        Erc20 underlying = Erc20(_erc20Contract);

        // Create a reference to the corresponding cToken contract, like cDAI
        CErc20 cToken = CErc20(_cErc20Contract);

        // Amount of current exchange rate from cToken to underlying
        uint256 exchangeRateMantissa = cToken.exchangeRateCurrent();
        emit MyLog("Exchange Rate (scaled up): ", exchangeRateMantissa);

        // Amount added to you supply balance this block
        uint256 supplyRateMantissa = cToken.supplyRatePerBlock();
        emit MyLog("Supply Rate: (scaled up)", supplyRateMantissa);

        // Approve transfer on the ERC20 contract
        underlying.approve(_cErc20Contract, _numTokensToSupply);

        // Mint cTokens
        uint mintResult = cToken.mint(_numTokensToSupply);
        return mintResult;
    }
```

The first function in **MyContract** allows the caller to supply an ERC20 token to the Compound Protocol. We will need to pass the underlying contract address, the cToken contract address, and the number of tokens we want to supply.

The function first creates references to the production instances of Dai and cDAI contracts using our interface definitions.

Then the function logs the **exchange rate** and the **supply rate**. These calls are not necessary for supplying. They are there for illustration. You can see the amounts in the “events” output later in JavaScript.

Next, our function approves the transfer of ERC20 token from our contract's address to Compound's cToken contract using the **approve** method.

Finally, our contract calls the cToken contract **mint** function. This supplies some Dai to the protocol, and gives our custom contract a balance of cDAI.

After we have supplied some Dai, we can redeem it at any time. The following function shows how we can accomplish that in Solidity.

```
contract MyContract {

    // omitting some code here...

    function redeemCErc20Tokens(
        uint256 amount,
        bool redeemType,
        address _cErc20Contract
    ) public returns (bool) {
        // Create a reference to the corresponding cToken contract, like cDAI
        CErc20 cToken = CErc20(_cErc20Contract);

        // `amount` is scaled up by 1e18 to avoid decimals

        uint256 redeemResult;

        if (redeemType == true) {
            // Retrieve your asset based on a cToken amount
            redeemResult = cToken.redeem(amount);
        } else {
            // Retrieve your asset based on an amount of the asset
            redeemResult = cToken.redeemUnderlying(amount);
        }

        // Error codes are listed here:
        // https://compound.finance/developers/ctokens#ctoken-error-codes
        emit MyLog("If this is not 0, there was an error", redeemResult);
        require(redeemResult == 0, "redeemResult error");

        return true;
    }
}
```

The **redeemCErc20Tokens** function allows the caller to redeem based on the amount of underlying or the amount of cTokens. This is indicated by calling the function with a boolean for redeem type; True for cToken, and false for underlying amount.

If there is an error with redeeming, the error code is logged using **MyLog**. Error codes for cToken contracts are described in the documentation.

Now that we have our code written, let's run it!

Compiling

I have written a script that compiles the Solidity code based on the version we indicated at the top of the file. If you **changed the solidity version**, the compile script **might not work**.

If you cloned the GitHub repository, be sure to run **npm install** in the root directory of the project before you try to run the compile script.

Compile your contract with the command below. The bytecode and ABI will be waiting in the **.build/** folder.

```
node compile-smart-contracts.js
```

Deploying

Once you have **deployed** your contract, the script will log the new **MyContract** address.

```
node deploy-smart-contracts.js
> Your contract was successfully deployed!
> The contract can be interfaced with at this address:
> 0x4a81cff73f1b8c6d94f50EDC08A4DEe7fbC109C6
```

Copy this and save it for later. We'll need it to call the smart contract's function to supply Dai to Compound.

Executing

The Web3.js code that will invoke our custom smart contract can be found in the **solidity-examples/** folder. Let's run through the **supply-erc20-via-solidity.js** script.

First, the script makes a Web3 object and points it to the blockchain network that we want to use to supply to Compound.

Next, we make a reference to our Ethereum wallet private key. This should be a wallet that has some ETH (for gas) and also Dai (to supply to Compound). Our script's main function first transfers Dai from our wallet to **MyContract**.

Next, we make some references to **MyContract**, the **Dai contract**, and also the **Compound cDAI contract**.

Remember, the cToken contract addresses and ABIs can be found here: https://compound.finance/docs#networks, and **MyContract's** address was **logged** when we deployed the contract.

See the **Minting Test Net DAI** section to find the newest Dai contract address.

```
/**
 * Executes our contract's `supplyErc20ToCompound` function
 *
 * Remember to run your local ganache-cli with the mnemonic so you have accounts
 * with ETH in your local Ethereum environment. Don't use the keys outside of
 * your local test environment.
 *
 * ./node_modules/.bin/ganache-cli \
 *     -f https://mainnet.infura.io/v3/<YOUR INFURA API KEY HERE> \
 *     -m "clutch captain shoe salt awake harvest setup primary inmate ugly among be
 *     -i 1 \
 *     -u 0x9759A6Ac90977b93B58547b4A71c78317f391A28
 */
const Web3 = require('web3');
const web3 = new Web3('http://127.0.0.1:8545');

// Set up a wallet using one of Ganache's key pairs.
// Don't use this key outside of your local test environment.
const privateKey = '0xb8c1b5c1d81f9475fdf2e334517d29f733bdfa40682207571b12fc1142cbf3

// Add your Ethereum wallet to the Web3 object
web3.eth.accounts.wallet.add(privateKey);
const myWalletAddress = web3.eth.accounts.wallet[0].address;

// `myContractAddress` is logged when running the deploy script in the root
// directory of the project. Run the deploy script prior to running this one.
const myContractAddress = '0x4a81cff73f1b8c6d94f50EDC08A4DEe7fbC109C6';
const myAbi = require('../.build/abi.json');
const myContract = new web3.eth.Contract(myAbi, myContractAddress);

// Mainnet address of the underlying token contract. Example: Dai.
// https://etherscan.io/address/0x6b175474e89094c44da98b954eedeac495271d0f
const daiMainNetAddress = '0x6b175474e89094c44da98b954eedeac495271d0f';
const daiAbi = [{"inputs":[{"internalType":"uint256","name":"chainId_","type":"uint2
const daiContract = new web3.eth.Contract(daiAbi, daiMainNetAddress);

// Mainnet contract address and ABI for the cToken, which can be found in the
// mainnet tab on this page: https://compound.finance/docs
const compoundCDaiContractAddress = '0x5d3a536e4d6dbd6114cc1ead35777bab948e3643';
const compoundCDaiContractAbi = [{"inputs":[{"internalType":"address","name":"underl
const compoundCDaiContract = new web3.eth.Contract(compoundCDaiContractAbi, compound
```

Finally, we call our main function, which first transfers Dai from our wallet to

MyContract.

```
const assetName = 'DAI'; // for the log output lines
const underlyingDecimals = 18; // Number of decimals defined in this ERC20 token's c

// Web3 transaction information, we'll use this for every transaction we'll send
const fromMyWallet = {
  from: myWalletAddress,
  gasLimit: web3.utils.toHex(500000),
  gasPrice: web3.utils.toHex(20000000000) // use ethgasstation.info (mainnet only)
};

const main = async function() {
  console.log(`Now transferring ${assetName} from my wallet to MyContract...`);

  let transferResult = await daiContract.methods.transfer(
    myContractAddress,
    web3.utils.toHex(10 * Math.pow(10, underlyingDecimals)) // 10 tokens to send to
  ).send(fromMyWallet);

  console.log(`MyContract now has ${assetName} to supply to the Compound Protocol.`)
```

supply-erc20-via-solidity.js hosted with ❤ by **GitHub** **view raw**

Next we call the **supplyErc20ToCompound** function in **MyContract**, which sends 10 Dai to Compound in exchange for cDAI.

Thanks for reading! You are now able to supply assets to the Compound protocol using Solidity or JavaScript. We walked through **supplying** Ether, and also the supported

```
// Mint some cDAI by sending DAI to the Compound Protocol
console.log(`MyContract is now minting c${assetName}...`);
let supplyResult = await myContract.methods.supplyErc20ToCompound(
  daiMainNetAddress,
  compoundCDaiContractAddress,
  web3.utils.toHex(10 * Math.pow(10, underlyingDecimals)) // 10 tokens to supply
).send(fromMyWallet);

console.log(`Supplied ${assetName} to Compound via MyContract`);
// Uncomment this to see the solidity logs
// console.log(supplyResult.events.MyLog);
```

supply-erc20-via-solidity.js hosted with ❤ by **GitHub** **view raw**

Ethereum Ethereum Development Solidity Tutorial Web3 Compound trate how to get the **balance of underlying** ERC20 asset in the protocol and the **amount of cTokens** that **MyContract** now holds.

```
let balanceOfUnderlying = await compoundCDaiContract.methods
  .balanceOfUnderlying(myContractAddress).call();
const balanceOfUnderlyingDai = web3.utils.fromWei(balanceOfUnderlying);
console.log(`${assetName} supplied to the Compound Protocol:`, balanceOfUnderlyingDai);

let cTokenBalance = await compoundCDaiContract.methods.balanceOf(myContractAddress).call()
cTokenBalance = cTokenBalance / 1e8;
console.log(`MyContract's c${assetName} Token Balance:`, cTokenBalance);
```

supply-erc20-via-solidity.js hosted with ❤ by **GitHub** **view raw**

Lastly we call the **redeemCErc20Tokens** function in MyContract to redeem the cDAI for Da
The example utilizes the **redeem** method by passing a cToken amount. Under that, there is a
redeem **underlying amount** example, which is commented out.

```
    // Call redeem based on a cToken amount
    const amount = web3.utils.toHex(cTokenBalance * 1e8);
    const redeemType = true; // true for `redeem`

    // Call redeemUnderlying based on an underlying amount
    // const amount = web3.utils.toHex(balanceOfUnderlying);
    // const redeemType = false; //false for `redeemUnderlying`

    // Retrieve your asset by exchanging cTokens
    console.log(`Redeeming the c${assetName} for ${assetName}...`);
    let redeemResult = await myContract.methods.redeemCErc20Tokens(
      amount,
      redeemType,
      compoundCDaiContractAddress
    ).send(fromMyWallet);

    if (redeemResult.events.MyLog.returnValues[1] != 0) {
      throw Error('Redeem Error Code: '+redeemResult.events.MyLog.returnValues[1]);
    }

    cTokenBalance = await compoundCDaiContract.methods.balanceOf(myContractAddress).call();
    cTokenBalance = cTokenBalance / 1e8;
    console.log(`MyContract's c${assetName} Token Balance:`, cTokenBalance);
  }

  main().catch((err) => {
    console.error(err);
  });
```

supply-erc20-via-solidity.js hosted with ❤ by **GitHub** **view raw**

Now we're ready to run!

If you are running on **localhost** remember to mint test DAI before you run the JavaScript fil

```
node seed-account-with-erc20/dai.js
```

If you are running this on a public network, you'll need to acquire Dai for that network.

To execute the script, navigate to the project root directory and run:

```
node solidity-examples/supply-erc20-via-solidity.js
```

If successful, the output of the script will show something like this:

```
$ node solidity-examples/supply-erc20-via-solidity.js
Now transferring DAI from my wallet to MyContract...
MyContract now has DAI to supply to the Compound Protocol.
MyContract is now minting cDAI...
Supplied DAI to Compound via MyContract
DAI supplied to the Compound Protocol: 9.999999999942267983
MyContract's cDAI Token Balance: 482.50440136
Redeeming the cDAI for DAI...
MyContract's cDAI Token Balance: 0
```

supply-erc20-via-solidity-output.txt hosted with ❤ by **GitHub** **view raw**

Remember that this code will work with any of the ERC20 tokens that Compound supports. You will need to swap in the corresponding ERC20 token contract address and ABI into the JavaScript.