Borrowing Assets from the Compound Protocol

Quick Start Guide

Adam Bavosa Follow
Mar 17, 2020 · 13 min read

The Compound Protocol allows users to borrow crypto assets, using any other supported asset as collateral — giving them the flexibility to settle a trade, or use an application, *with an asset that they don't already own*.

For example, a user holding ETH may supply it to Compound and borrow DAI from Compound instantly.

This guide will teach you how to develop applications that facilitate borrowing assets from the Compound protocol using **JavaScript** and **Solidity**.

What drives the interest that protocol suppliers

earn?

Borrowers. Those that borrow crypto assets from the Compound protocol pay a varying interest rate every Ethereum block. *The interest that borrowers pay produces the interest that suppliers earn.*

Before continuing, be sure to read the guide for Supplying Assets to the Compound Protocol.

Table of Contents for This Guide

How Do I Borrow Assets From Compound?

Borrowing can be done with a user interface, or with **code**. Before we walk through the steps of a borrowing sequence, let's cover some **key concepts**.

1. **Collateral** — In order to borrow crypto from the Compound protocol, users need to first supply another type of crypto as collateral. This is provided using the same **mint** function used for supplying assets. Supplied collateral assets earn interest while in the protocol, but users cannot redeem or transfer assets while they are being used as collateral.

2. **Collateral Factor** — The maximum amount users can borrow is limited by the collateral factors of the assets they have supplied. For example, if a user supplies 100 DAI as collateral, and the posted collateral factor for DAI is 75%, then the user can borrow at most 75 DAI worth of other assets at any given time. Each asset on Compound can have a different collateral factor. Collateral factors for each asset can be fetched using the Comptroller contract. They can also be changed by Compound Governance, with a minimum waiting period of five days.

3. **Enter Markets** — An asset that is supplied to the protocol is not usable as collateral initially. In order to inform the protocol that you wish to use an asset as collateral, you must "enter the market" for that asset. An account can enter multiple markets at one time.

4. **Calculating Account Liquidity** — How can you tell what your maximum allowed borrow amount is when you have multiple collateral assets with differing collateral factors? The Comptroller contract provides an easy to use function that calculates your account's liquidity, which is a USD-denominated value of the maximum allowed borrow amount. You should never borrow this much at once because your account would instantly be liquidated as soon as the protocol's "accrue interest" operation is executed.

5. **The Open Price Feed** — How can the protocol calculate the maximum borrow, or if a user's account is insolvent? Compound has its own Price Feed contract that has the current exchange rates of all supported assets. The Open Price Feed uses price feeds from several highly liquid exchanges to find the median price, and has rules coded into the contract to ensure the integrity of its reported prices. You can view the Open Price Feed documentation on the Compound website.

6. **The Borrow Balance** — This is the sum of a user's current borrowed amount plus the interest that needs to be repaid. This is calculated with an easy to use function in each cToken contract.

7. **The Borrow Rate** — Borrowers owe the prevailing interest rate of the asset they are borrowing. This is done by adding the Borrow Rate to the account's **Borrow Balance** every Ethereum block. While a borrow is open, the Borrow Balance is **ever-increasing**. The borrow and interest is never required to be repaid unless the borrower becomes insolvent; otherwise, the borrower can choose to repay some or all of their borrow whenever they choose.

8. **Liquidation** — A borrowing account becomes insolvent when the Borrow Balance exceeds the amount allowed by the collateral factor. When an account becomes insolvent, other users can repay a portion of its outstanding borrow in exchange for a portion of its collateral, with a liquidation incentive — currently set at 8% but subject to change through Compound's governance system. The liquidation incentive means that liquidators receive the borrower's collateral at a 8% discount to market price. Having your account liquidated is **bad** because you lose some of your collateral.

9. **Repaying a Borrow** — Borrows can be repaid using a function on the respective cToken contract. Once a borrow has been repaid, the account's collateral can be entirely redeemed or transferred. There are also functions in the cToken contracts to repay a borrow on behalf of another account.

Let's dive into borrowing examples using **JavaScript** or **Solidity**. The order of operations for borrowing are:

Supply Collateral

1. Supply one or many supported crypto assets to Compound, as **collateral**, by calling the **mint** function in the cToken smart contract. You will receive cTokens in return for your asset. Note that collateral earns interest, even while a borrow is open against it.

2. Enter the market of the supplied asset or assets by calling the Comptroller's **enterMarkets** function. Call this function and pass the address of the cToken contract of the asset in which you want to use as **collateral**.

3. Choose an asset you want to **borrow** from the supported assets listed on the Compound Markets page. All cToken **contract addresses** are posted in the Compound documentation.

Calculate the amount to borrow

1. Use the Open Price Feed contract's **price** function to get the price in USD of the asset you wish to borrow. We'll need this to find out just how much we can borrow.

2. Call **getAccountLiquidity** on the Comptroller to get the USD value of your account's liquidity.

3. Calculate the maximum amount you can borrow of the asset. Divide the account's liquidity by the price of the asset you wish to borrow. This step is not needed when you are borrowing USDC or USDT because the liquidity price is already in USD.

Borrow

1. Call the corresponding cToken contract's **borrow** function, passing an amount that is less than your maximum allowed borrow.

2. Do things with your borrowed asset while making sure your account does not go **insolvent**.

3. When you're ready, repay your borrow using the cToken's **repayBorrow** function.

This can be done solely in **JavaScript** using Web3.js JSON RPC or with a proxy smart contract written in **Solidity**.

Testing Your Borrow Code

If you aren't familiar with localhost testing with Ganache-CLI, take a look at the development environment quick start guide to learn more about it, and install it.

We're going to fork the Ethereum main network using this command. We are using an Ethereum wallet mnemonic to generate our address and private keys. **Do not use this wallet outside of your local test environment!** Be sure to supplant your Infura project ID in the JSON RPC provider URL.

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

Leave Ganache CLI running in the background while you run the code examples.

We are unlocking an address that will allow us to mint DAI for the examples in which we supply DAI. When you are ready to run this code in production, you will have to acquire your own DAI on the Ethereum Mainnet.

Compound Borrow Example Code on GitHub

All of the code referenced in this guide is available in the Compound Borrow Examples repository on GitHub. Please **clone this repository to your local machine** and follow along with the examples.

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

Be sure to navigate to the root directory of the cloned repo on your command line. Then install all of the dependencies listed in **package.json** using the following command.

```
cd compound-borrow-examples/
npm install
## or: yarn install
```

Borrowing an ERC20 Token Using ETH as Collateral — JavaScript

Let's supply Ether to Compound and borrow Dai, an ERC20 stablecoin. This code will work for borrowing all other ERC20 assets using Ether. Obviously, you'll need to replace the Dai and cDai references with the token you wish to borrow.

If you haven't already, acquire some Ether and open your code editor. You don't need to acquire Ether if you are testing locally with Ganache CLI.

We'll be using Node.js JavaScript with Web3.js JSON RPC to call the Compound protocol's smart contracts. This JavaScript code will work in the web browser too, but you need to import the Web3 library using a **script tag** instead of **require**.

First, let's use **Web3.js** to connect to the Ethereum network nodes. Replace the endpoint with a different Ethereum provider URL if you are ready to use a public blockchain instead of Ganache CLI.

Next we'll import some contract addresses and our Ethereum wallet's private key. The JSON file referenced is available in the GitHub Repository. I'll show the private key as a string literal, but it's a best practice to import it using an **environment variable**.

Each of the Ethereum addresses and ABIs are available online in the Compound Protocol Documentation.

For web browser JavaScript, you will need to use Browserify to import the configuration using **require**. Alternatively, you can copy and paste each of the variables into your code file.

Next, we'll make some contract objects using Web3.js. This makes it simple to interact with the smart contracts on our sandbox test blockchain. We'll also add some supplantable asset information and transaction parameters.

Next, we'll make a log function, which we will execute several times throughout the borrow code process. This will illustrate our balances that change as we execute the Compound smart contract methods.

Finally we make our async main function, its call, and error handler.

Here is a list of the steps in their order of execution:

- Supply 1 ETH to Compound, which will become our collateral.
- Enter the ETH market using the comptroller to tell the protocol that we want to use the ETH as collateral.
- Get the collateral factor for our collateral asset (in this case ETH) using the comptroller.
- Get the Dai price in ETH using the price oracle.

- Calculate our maximum Dai borrow.
- Calculate the per-block interest rate applied to borrows.
- Borrow 50 DAI.
- Get our account's borrow balance.
- Repay the borrow in full.

All of the code for this JS file is available in the borrow example repository. Let's try to execute it. From the root directory of the borrow examples repository, run the following command line command.

```
  node web3-js-examples/borrow-erc20-with-eth-collateral.js
```

If you are still testing locally, remember to run Ganache CLI in the background using the command above before you execute the JSON RPC script.

You should see something similar to the following output on your command line.

When you borrow assets from the Compound Protocol, you earn interest on your collateral. However, you cannot transfer or redeem the collateral cTokens that are supporting an open borrow.

Remember to repay all or some of your borrow **prior** to redeeming/transferring cTokens.

Borrowing ETH Using an ERC20 Token as Collateral — Solidity

Let's supply Dai to the protocol as collateral and borrow ETH using Solidity. This code will work for supplying any other supported ERC20 token as collateral. You would need to change the references and the contract addresses in the code.

We'll deploy a Solidity smart contract that will receive a Dai balance from our wallet. Next the contract supplies the Dai to compound on its own behalf, before borrowing ETH. The ETH borrow balance will be held by the contract itself.

If you haven't already, acquire some Dai and open your code editor. You don't need to acquire Ether if you are testing locally with Ganache CLI.

We'll be invoking our smart contract using Web3.js. More on that later. Let's go through writing **MyContracts.sol** which will be our Solidity code file.

Solidity Code

First, we select our version of Solidity, and make some Solidity interface definitions. These are used to call other deployed contracts, without having to add their entire codebase to ours. We need to reference the Dai contract, so we'll make an ERC20 wrapper.

Let’s create our contract and call it **MyContract**. We’ll add a function for borrowing ETH. It will need some contract addresses passed to it as parameters.

Now that we have declared our references to external contracts needed in this operation, we'll approve a transfer of Dai from this contract to the protocol. After the approve operation, we can mint some cDai. This is necessary when supplying Dai as collateral.

Next, we “enter” the Dai market. This is the way we indicate that we want to use our underlying Dai as collateral, so we can borrow different assets from the protocol.

Before we borrow ETH, we need to determine the maximum amount of ETH we can borrow. This is important because if we try to borrow more than we are allowed to, the operation will fail. Also, if we borrow too close to the limit, our account will be liquidated.

This is done by calling the **getAccountLiquidity** function on the protocol's comptroller contract. The amount gets logged by our Solidity event, so the Web3.js return value will show the amount in its event logs.

Next I have some code that logs things like the **maximum borrow**, the **collateral factor** and the **borrow rate** per block. These are not necessary for the execution, but I wanted to illustrate how you would use them if needed.

The last step in the borrow operation is to pass the amount of ETH that we want to borrow to the cToken contract's borrow function. We are also going to fetch how much we have borrowed using the **borrowBalanceCurrent** method on the cToken contract. We'll return this value.

There is one more step to making this function work. Remember that our contract needs to receive ETH once we call the borrow function. The **borrowEthExample** we just wrote is not a payable function.

We need to add a **payable** function to the contract. In Solidity version 0.5.12, the payable fallback function looks like this.

Now when we execute the **borrowEthExample** function, our fallback function will accept the borrowed Ether that is transferred from the protocol.

Repaying The Borrow

Finally, let’s add some code to **repay** our borrow. This is a separate function. We’ll call this function after we call **borrowEthExample**.

Compiling and Deploying the Contract

If you are new to deploying contracts, or need a refresher, please read **Setting Up an Ethereum Development Environment** quick start guide.

To deploy, we'll use compile-smart-contracts.js script and deploy-smart-contracts.js. Make sure Ganache CLI is running using the command from earlier, if you are still testing locally.

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

```
> 0x8dF3b210283F08eC30da4e8fF8bf62981FbBef34
```

Save this address for later! We'll need to reference the contract address from our Web3.js code.

Web3 JavaScript Code

In order to execute our solidity smart contract code, we need to write some JavaScript to call it via JSON RPC. This is done with Web3.js. The following code is available in **borrow-eth-via-solidity.js** in the GitHub repository.

If you haven't already, acquire some Dai and open your code editor. To acquire Dai on your localhost test net fork of the Ethereum mainnet, execute the seed script while Ganache CLI is running in another command line window. There are instructions in the GitHub repository readme.

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

First, let's use web3.js to connect to the blockchain just like we did in the previous section, **Borrowing an ERC20 Token Using ETH as Collateral — JavaScript**. Replace the endpoint with a different Ethereum network URL if you are ready to use a public blockchain instead of Ganache CLI.

Import the contract addresses and our Ethereum wallet's private key. The JSON file is available in the GitHub Repository.

Be sure to **paste the contract address of MyContract** in the address variable value. This address was logged when you deployed (or re-deployed) the contract (see above **Compiling and Deploying the Contract**).

We'll make a similar log function, to illustrate our balances that change as we execute the Compound smart contract methods.

The main function transfers Dai from your wallet to **MyContract**. Then it calls the **borrowEthExample** function in **MyContract** via Web3.js JSON RPC. Lastly, we call **myEthRepayBorrow** to repay the contract's borrow.

Here is the order of events that occur.

vallet to **MyContract**.

- Call **borrowEthExample** using Web3.js JSON RPC.
- Approve 25 Dai to be moved from **MyContract** to the Compound protocol.
- Supply 25 Dai to the protocol, which will become our collateral.
- Enter the Dai market using the comptroller to designate Dai as collateral.
- Get our total liquid assets in the protocol using the comptroller.
- Get the maximum amount of ETH in **Wei** that we can borrow from the protocol.
- Borrow 0.02 ETH.
- Check the Borrow Balance to see how much ETH we are currently borrowing. This is returned by the function.
- Repay the borrow in full using the **myEthRepayBorrow** function.

Here is our async main function, its call, and error handler.

All of the code for this JS file is available in the borrow example repository. Let's try to execu it.

Remember to get some Dai before you try running this script. If you are testing on your localhost using Ganache CLI, execute the **seed Dai script** using the command **described earlier**.

From the root directory of the borrow examples repository, run the following command line command.

```
node solidity-examples/borrow-eth-via-solidity.js
```

You should see something similar to the following output on your command line.

Thank you for following along with this guide for borrowing assets from the Compound protocol with JavaScript and Solidity. More examples for borrowing and redeeming of crypt assets are available in the Compound Borrow Examples GitHub repository.

In the future, we can go over implementing practical applications of this code in your DApp, you can build something like the products featured on the Compound home page.

For more updates and insights from companies that implement the Compound protocol, be sure to subscribe to the Compound Newsletter. If you have questions or would like help, **joir us** in the #development channel of the Compound Discord.