---

title: Calling a smart contract from JavaScript description: How to call a smart contract function from JavaScript using a Dai token example author: jdourlens tags: ["transactions", "frontend", "JavaScript", "web3.js"] skill: beginner lang: en published: 2020-04-19 source: EthereumDev sourceUrl: https://ethereumdev.io/calling-a-smart-contract-from-javascript/ address: "0x19dE91Af973F404EDF5B4c093983a7c6E3EC8ccE"

In this tutorial we'll see how to call a smart contract function from JavaScript. First is reading the state of a smart contract (e.g. the balance of an ERC20 holder), then we'll modify the state of the blockchain by making a token transfer. You should already be familiar with setting up a JS environment to interact with the blockchain

For this example we'll play with the DAI token, for testing purpose we'll fork the blockchain using ganache-cli and unlock an address that already has a lot of DAI:

```
bash ganache-cli -f https://mainnet.infura.io/v3/[YOUR INFURA KEY] -d -i 66 1 --unlock
0x4d10ae710Bd8D1C31bd7465c8CBC3add6F279E81
```

To interact with a smart contract we'll need its address and ABI:

\`\`\`js const ERC20TransferABI = [{ constant: false, inputs: [{ name: "_to", type: "address", }, { name: "_value", type: "uint256", },], name: "transfer", outputs: [{ name: "", type: "bool", },], payable: false, stateMutability: "nonpayable", type: "function", }, { constant: true, inputs: [{ name: "_owner", type: "address", },], name: "balanceOf", outputs: [{ name: "balance", type: "uint256", },], payable: false, stateMutability: "view", type: "function", },]

const DAI_ADDRESS = "0x6b175474e89094c44da98b954eedeac495271d0f" \`\`\`

For this project we stripped the complete ERC20 ABI to just keep the `balanceOf` and `transfer` function but you can find the full ERC20 ABI here.

We then need to instantiate our smart contract:

\`\`\`js const web3 = new Web3("http://localhost:8545")

const daiToken = new web3.eth.Contract(ERC20TransferABI, DAI_ADDRESS) \`\`\`

We'll also set up two addresses:

- the one who will receive the transfer and
- the one we already unlocked that will send it:

```
js const senderAddress = "0x4d10ae710Bd8D1C31bd7465c8CBC3add6F279E81" const receiverAddress =
"0x19dE91Af973F404EDF5B4c093983a7c6E3EC8ccE"
```

In the next part we'll call the `balanceOf` function to retrieve the current amount of tokens both addresses hold.

Call: Reading value from a smart contract {#call-reading-value-from-a-smart-contract}

The first example will call a "constant" method and execute its smart contract method in the EVM without sending any transaction. For this we'll read the ERC20 balance of an address. Read our article about ERC20 tokens.

You can access an instantiated smart contract methods that you provided the ABI for as follows: `yourContract.methods.methodname`. By using the `call` function you'll receive the result of executing the function.

```
js daiToken.methods.balanceOf(senderAddress).call(function (err, res) { if (err) { console.log("An error
occurred", err) return } console.log("The balance is: ", res) })
```

Remember that DAI ERC20 has 18 decimals which means you need to remove 18 zeros to get the correct amount. uint256 are returned as strings as JavaScript does not handle big numeric values. If you're not sure how to deal with big numbers in JS check our tutorial about bignumber.js.

Send: Sending a transaction to a smart contract function {#send-sending-a-transaction-to-a-smart-contract-function}

For the second example we’ll call the transfer function of the DAI smart contract to send 10 DAI to our second address. The transfer function accepts two parameters: the recipient address and the amount of token to transfers:

```
js daiToken.methods .transfer(receiverAddress, "100000000000000000000") .send({ from: senderAddress }, function
(err, res) { if (err) { console.log("An error occurred", err) return } console.log("Hash of the transaction: " +
res) })
```

The call function returns the hash of the transaction that will be mined into the blockchain. On Ethereum, transaction hashes are predictable - that’s how we can get the hash of the transaction before it is executed (learn how hashes are calculated here).

As the function only submits the transaction to the blockchain, we can’t see the result until we know when it is mined and included in the blockchain. In the next tutorial we’ll learn how to wait for a transaction to be executed on the blockchain by knowing its hash.