title: Hello World Smart Contract for Beginners - Fullstack description: Introductory tutorial on writing and deploying a simple smart contract on Ethereum. author: "nstrike2" tags: ["solidity", "hardhat", "alchemy", "smart contracts", "deploying", "blockexplorer", "frontend", "transactions",] skill: beginner lang: en published: 2021-10-25

This guide is for you if you are new to blockchain development and don't know where to start or how to deploy and interact with smart contracts. We will walk through creating and deploying a simple, smart contract on the Goerli test network using MetaMask, Solidity, Hardhat, and Alchemy.

You'll need an Alchemy account to complete this tutorial.Sign up for a free account.

If you have questions at any point, feel free to reach out in theAlchemy Discord!

Part 1 - Create and Deploy your Smart Contract using Hardhat {#part-1}

Connect to the Ethereum network {#connect-to-the-ethereum-network}

There are many ways to make requests to the Ethereum chain. For simplicity, we'll use a free account on Alchemy, a blockchain developer platform and API that allows us to communicate with the Ethereum chain without running a node ourselves. Alchemy also has developer tools for monitoring and analytics; we'll take advantage of these in this tutorial to understand what's going on under the hood in our smart contract deployment.

Create your app and API key {#create-your-app-and-api-key}

Once you've created an Alchemy account, you can generate an API key by creating an app. This will allow you to make requests to the Goerli testnet. If you're not familiar with testnets you can read Alchemy's guide to choosing a network

On the Alchemy dashboard, find the**Apps** dropdown in the navigation bar and click**Create App**.

Give your app the name *'Hello World'* and write a short description. Select**Staging** as your environment and**Goerli** as your network.

*Note: be sure to select **Goerli**, or this tutorial won't work.*

Click **Create app**. Your app will appear in the table below.

Create an Ethereum account {#create-an-ethereum-account}

You need an Ethereum account to send and receive transactions. We'll use MetaMask, a virtual wallet in the browser that lets users manage their Ethereum account address.

You can download and create a MetaMask account for freehere. When you are creating an account, or if you already have an account, make sure to switch over to the “Goerli Test Network” in the upper right (so that we’re not dealing with real money).

Step 4: Add ether from a Faucet {#step-4-add-ether-from-a-faucet}

To deploy your smart contract to the test network, you'll need some fake ETH. To get ETH on the Goerli network, go to a Goerli faucet and enter your Goerli account address. Note that Goerli faucets can be a bit unreliable recently - see the test networks page for a list of options to try:

Note: due to network congestion, this might take a while.``

Step 5: Check your Balance {#step-5-check-your-balance}

To double-check the ETH is in your wallet, let’s make aneth_getBalance request using Alchemy’s composer tool. This will return the amount of ETH in our wallet. To learn more check out Alchemy's short tutorial on how to use the composer tool

Enter you input your MetaMask account address and click**Send Request**. You will see a response that looks like the code snippet below.

`json { "jsonrpc": "2.0", "id": 0, "result": "0x2B5E3AF16B1880000" }`

> *Note: This result is in wei, not ETH. Wei is used as the smallest denomination of ether.*

Phew! Our fake money is all there.

Step 6: Initialize our project {#step-6-initialize-our-project}

First, we'll need to create a folder for our project. Navigate to your command line and input the following.

`mkdir hello-world cd hello-world`

Now that we're inside our project folder, we'll use`npm init` to initialize the project.

> If you don't have npm installed yet, follow these instructions to install Node.js and npm.

For the purpose of this tutorial, it doesn't matter how you answer the initialization questions. Here is how we did it for reference:

``` package name: (hello-world) version: (1.0.0) description: hello world smart contract entry point: (index.js) test command: git repository: keywords: author: license: (ISC)

About to write to /Users/.../.../.../hello-world/package.json:

{ "name": "hello-world", "version": "1.0.0", "description": "hello world smart contract", "main": "index.js", "scripts": { "test": "echo \"Error: no test specified\" && exit 1" }, "author": "", "license": "ISC" } ```

Approve the package.json and we're good to go!

## Step 7: Download Hardhat {#step-7-download-hardhat}

Hardhat is a development environment to compile, deploy, test, and debug your Ethereum software. It helps developers when building smart contracts and dapps locally before deploying to the live chain.

Inside our `hello-world` project run:

`npm install --save-dev hardhat`

Check out this page for more details oninstallation instructions.

## Step 8: Create Hardhat project {#step-8-create-hardhat-project}

Inside our `hello-world` project folder, run:

`npx hardhat`

You should then see a welcome message and option to select what you want to do. Select "create an empty hardhat.config.js":

``` 888 888 888 888 888 888 888 888 888 888 888 888 888 888 888 8888888888 8888b. 888d888 .d88888 88888b. 8888b. 888888 888 888 "88b 888P" d88" 888 888 "88b "88b 888 888 888 .d888888 888 888 888 888 888 .d888888 888 888 888 888 888 888 Y88b 888 888 888 888 888 Y88b. 888 888 "Y888888 888 "Y88888 888 888 "Y888888 "Y888

Welcome to Hardhat v2.0.11

What do you want to do? … Create a sample project ❯ Create an empty hardhat.config.js Quit ```

This will generate a `hardhat.config.js` file in the project. We'll use this later in the tutorial to specify the setup for our project.

## Step 9: Add project folders {#step-9-add-project-folders}

To keep the project organized, let's create two new folders. In the command line, navigate to the root directory of your `hello-world` project and type:

`mkdir contracts mkdir scripts`

- `contracts/` is where we'll keep our hello world smart contract code file

- `scripts/` is where we'll keep scripts to deploy and interact with our contract

## Step 10: Write our contract {#step-10-write-our-contract}

You might be asking yourself, when are we going to write code? It's time!

Open up the hello-world project in your favorite editor. Smart contracts most commonly are written in Solidity, which we will use to write our smart contract.

1. Navigate to the `contracts` folder and create a new file called`HelloWorld.sol`
2. Below is a sample Hello World smart contract that we will be using for this tutorial. Copy the contents below into the `HelloWorld.sol` file.

*Note: Be sure to read the comments to understand what this contract does.*

``` // Specifies the version of Solidity, using semantic versioning. // Learn more: https://solidity.readthedocs.io/en/v0.5.10/layout-of-source-files.html#pragma pragma solidity >=0.7.3;

// Defines a contract named `HelloWorld`. // A contract is a collection of functions and data (its state). Once deployed, a contract resides at a specific address on the Ethereum blockchain. Learn more: https://solidity.readthedocs.io/en/v0.5.10/structure-of-a-contract.html contract HelloWorld {

//Emitted when update function is called //Smart contract events are a way for your contract to communicate that something happened on the blockchain to your app front-end, which can be 'listening' for certain events and take action when they happen. event UpdatedMessages(string oldStr, string newStr);

// Declares a state variable `message` of type `string`. // State variables are variables whose values are permanently stored in contract storage. The keyword `public` makes variables accessible from outside a contract and creates a function that other contracts or clients can call to access the value. string public message;

// Similar to many class-based object-oriented languages, a constructor is a special function that is only executed upon contract creation. // Constructors are used to initialize the contract's data. Learn more:https://solidity.readthedocs.io/en/v0.5.10/contracts.html#constructors constructor(string memory initMessage) {

```
  // Accepts a string argument `initMessage` and sets the value into the contract's `message` storage variable).
  message = initMessage;
```

}

// A public function that accepts a string argument and updates the`message` storage variable. function update(string memory newMessage) public { string memory oldMsg = message; message = newMessage; emit UpdatedMessages(oldMsg, newMessage); } } ```

This is a basic smart contract that stores a message upon creation. It can be updated by calling the`update` function.

Step 11: Connect MetaMask & Alchemy to your project {#step-11-connect-metamask-alchemy-to-your-project}

We've created a MetaMask wallet, Alchemy account, and written our smart contract, now it's time to connect the three.

Every transaction sent from your wallet requires a signature using your unique private key. To provide our program with this permission, we can safely store our private key in an environment file. We will also store an API key for Alchemy here.

> To learn more about sending transactions, check out this tutorial on sending transactions using web3.

First, install the dotenv package in your project directory:

```
npm install dotenv --save
```

Then, create a `.env` file in the root directory of the project. Add your MetaMask private key and HTTP Alchemy API URL to it.

Your environment file must be named `.env` or it won't be recognized as an environment file.

Do not name it`process.env` or `.env-custom` or anything else.
```

- Follow these instructions to export your private key
- See below to get HTTP Alchemy API URL

Your `.env` should look like this:

```
API_URL = "https://eth-goerli.alchemyapi.io/v2/your-api-key" PRIVATE_KEY = "your-metamask-private-key"
```

To actually connect these to our code, we'll reference these variables in our`hardhat.config.js` file on step 13.

Step 12: Install Ethers.js {#step-12-install-ethersjs}

Ethers.js is a library that makes it easier to interact and make requests to Ethereum by wrappingstandard JSON-RPC methods with more user friendly methods.

Hardhat allows us to integrateplugins for additional tooling and extended functionality. We'll be taking advantage of theEthers plugin for contract deployment.

In your project directory type:

```
bash npm install --save-dev @nomiclabs/hardhat-ethers "ethers@^5.0.0"
```

Step 13: Update hardhat.config.js {#step-13-update-hardhat.configjs}

We've added several dependencies and plugins so far, now we need to update`hardhat.config.js` so that our project knows about all of them.

Update your `hardhat.config.js` to look like this:

```javascript /* * *@type import('hardhat/config').HardhatUserConfig /*

require("dotenv").config() require("@nomiclabs/hardhat-ethers")

const { API_URL, PRIVATE_KEY } = process.env

module.exports = { solidity: "0.7.3", defaultNetwork: "goerli", networks: { hardhat: {}, goerli: { url: API_URL, accounts: [`0x${PRIVATE_KEY}`], }, }, } ```

### Step 14: Compile our contract {#step-14-compile-our-contract}

To make sure everything is working so far, let's compile our contract. The`compile` task is one of the built-in hardhat tasks.

From the command line run:

```
bash npx hardhat compile
~~~

You might get a warning about`SPDX license identifier not provided in source file`, but no need to worry about that — hopefully everything else looks good! If not, you can always message in the Alchemy discord.

Step 15: Write our deploy script {#step-15-write-our-deploy-script}

Now that our contract is written and our configuration file is good to go, it's time to write our contract deploy script.

Navigate to the `scripts/` folder and create a new file called`deploy.js` , adding the following contents to it:

```javascript async function main() { const HelloWorld = await ethers.getContractFactory("HelloWorld")

// Start deployment, returning a promise that resolves to a contract object const hello_world = await HelloWorld.deploy("Hello World!") console.log("Contract deployed to address:", hello_world.address) }

main() .then(() => process.exit(0)) .catch((error) => { console.error(error) process.exit(1) }) ```

Hardhat does an amazing job of explaining what each of these lines of code does in theirContracts tutorial, we've adopted their explanations here.

~~~
javascript const HelloWorld = await ethers.getContractFactory("HelloWorld")
```
```

A `ContractFactory` in ethers.js is an abstraction used to deploy new smart contracts, so `HelloWorld` here is a factory for instances of our hello world contract. When using the `hardhat-ethers` plugin `ContractFactory` and `Contract`, instances are connected to the first signer (owner) by default.

```
javascript const hello_world = await HelloWorld.deploy()
```

Calling `deploy()` on a `ContractFactory` will start the deployment, and return a `Promise` that resolves to a `Contract` object. This is the object that has a method for each of our smart contract functions.

Step 16: Deploy our contract {#step-16-deploy-our-contract}

We're finally ready to deploy our smart contract! Navigate to the command line and run:

```
bash npx hardhat run scripts/deploy.js --network goerli
```

You should then see something like:

```
bash Contract deployed to address: 0x6cd7d44516a20882cEa2DE9f205bF401c0d23570
```

Please save this address. We will be using it later in the tutorial.

If we go to the Goerli etherscan and search for our contract address we should able to see that it has been deployed successfully. The transaction will look something like this:

The `From` address should match your MetaMask account address and the `To` address will say **Contract Creation**. If we click into the transaction we'll see our contract address in the `To` field.

Congrats! You just deployed a smart contract to an Ethereum testnet.

To understand what's going on under the hood, let's navigate to the Explorer tab in our Alchemy dashboard. If you have multiple Alchemy apps make sure to filter by app and select **Hello World**.

Here you'll see a handful of JSON-RPC methods that Hardhat/Ethers made under the hood for us when we called the `.deploy()` function. Two important methods here are `eth_sendRawTransaction`, which is the request to write our contract onto the Goerli chain, and `eth_getTransactionByHash`, which is a request to read information about our transaction given the hash. To learn more about sending transactions, check out our tutorial on sending transactions using Web3.

Part 2: Interact with your Smart Contract {#part-2-interact-with-your-smart-contract}

Now that we've successfully deployed a smart contract to the Goerli network let's learn how to interact with it.

Create a interact.js file {#create-a-interactjs-file}

This is the file where we'll write our interaction script. We'll be using the Ethers.js library that you previously installed in Part 1.

Inside the `scripts/` folder, create a new file named `interact.js` add the following code:

```javascript // interact.js

const API_KEY = process.env.API_KEY const PRIVATE_KEY = process.env.PRIVATE_KEY const CONTRACT_ADDRESS = process.env.CONTRACT_ADDRESS ```

### Update your .env file {#update-your-env-file}

We will be using new environment variables, so we need to define them in the `.env` file that we created earlier.

We'll need to add a definition for our Alchemy `API_KEY` and the `CONTRACT_ADDRESS` where your smart contract was deployed.

Your `.env` file should look something like this:
```

~~~bash

# .env

API_URL = "https://eth-goerli.alchemyapi.io/v2/" API_KEY = "" PRIVATE_KEY = "" CONTRACT_ADDRESS = "0x" ```

### Grab your contract ABI {#grab-your-contract-ABI}

Our contract ABI (Application Binary Interface) is the interface to interact with our smart contract. Hardhat automatically generates an ABI and saves it in `HelloWorld.json`. To use the ABI, we'll need to parse out the contents by adding the following lines of code to our `interact.js` file:

`javascript // interact.js const contract = require("../artifacts/contracts/HelloWorld.sol/HelloWorld.json")`

If you want to see the ABI you can print it to your console:

`javascript console.log(JSON.stringify(contract.abi))`

To see your ABI printed to the console, navigate to your terminal and run:

`bash npx hardhat run scripts/interact.js`

### Create an instance of your contract {#create-an-instance-of-your-contract}

To interact with our contract, we need to create a contract instance in our code. To do so with Ethers.js, we'll need to work with three concepts:

1. Provider - a node provider that gives you read and write access to the blockchain
2. Signer - represents an Ethereum account that can sign transactions
3. Contract - an Ethers.js object representing a specific contract deployed on-chain

We'll use the contract ABI from the previous step to create our instance of the contract:

```javascript // interact.js

// Provider const alchemyProvider = new ethers.providers.AlchemyProvider( (network = "goerli"), API_KEY )

// Signer const signer = new ethers.Wallet(PRIVATE_KEY, alchemyProvider)

// Contract const helloWorldContract = new ethers.Contract( CONTRACT_ADDRESS, contract.abi, signer ) ```

Learn more about Providers, Signers, and Contracts in theethers.js documentation.

### Read the init message {#read-the-init-message}

Remember when we deployed our contract with the`initMessage = "Hello world!"`? We are now going to read that message stored in our smart contract and print it to the console.

In JavaScript, asynchronous functions get used when interacting with networks. To learn more about asynchronous functions,read this medium article.

Use the code below to call the`message` function in our smart contract and read the init message:

```javascript // interact.js

// ...

async function main() { const message = await helloWorldContract.message() console.log("The message is: " + message) } main() ```

After running the file using`npx hardhat run scripts/interact.js` in the terminal we should see this response:

~~~
The message is: Hello world!
```

Congrats! You've just successfully read smart contract data from the Ethereum blockchain, way to go!
```

### Update the message {#update-the-message}

Instead of just reading the message, we can also update the message saved in our smart contract using the `update` function! Pretty cool, right?

To update the message, we can directly call the `update` function on our instantiated Contract object:

```javascript // interact.js

// ...

async function main() { const message = await helloWorldContract.message() console.log("The message is: " + message)

console.log("Updating the message...") const tx = await helloWorldContract.update("This is the new message.") await tx.wait() } main() ```

Note that on line 11, we make a call to `.wait()` on the returned transaction object. This ensures that our script waits for the transaction to get mined on the blockchain before exiting the function. If the `.wait()` call isn't included, the script may not see the updated `message` value in the contract.

### Read the new message {#read-the-new-message}

You should be able to repeat the previous step to read the updated `message` value. Take a moment and see if you can make the changes necessary to print out that new value!

If you need a hint, here's what your `interact.js` file should look like at this point:

```javascript // interact.js

const API_KEY = process.env.API_KEY const PRIVATE_KEY = process.env.PRIVATE_KEY const CONTRACT_ADDRESS = process.env.CONTRACT_ADDRESS

const contract = require("../artifacts/contracts/HelloWorld.sol/HelloWorld.json")

// provider - Alchemy const alchemyProvider = new ethers.providers.AlchemyProvider( (network = "goerli"), API_KEY )

// signer - you const signer = new ethers.Wallet(PRIVATE_KEY, alchemyProvider)

// contract instance const helloWorldContract = new ethers.Contract( CONTRACT_ADDRESS, contract.abi, signer )

async function main() { const message = await helloWorldContract.message() console.log("The message is: " + message)

console.log("Updating the message...") const tx = await helloWorldContract.update("this is the new message") await tx.wait()

const newMessage = await helloWorldContract.message() console.log("The new message is: " + newMessage) }

main() ```

Now just run the script and you should be able to see the old message, the updating status, and the new message printed out to your terminal!

```
npx hardhat run scripts/interact.js --network goerli
```

```
The message is: Hello World! Updating the message... The new message is: This is the new message.
```

While running that script, you may notice that the `Updating the message...` step takes a while to load before the new message loads. That is due to the mining process; if you are curious about tracking transactions while they are being mined, visit the Alchemy mempool to see the status of a transaction. If the transaction is dropped, it's also helpful to check Goerli Etherscan and search for your transaction hash.

Part 3: Publish your Smart Contract to Etherscan {#part-3-publish-your-smart-contract-to-etherscan}

You did all the hard work of bringing your smart contract to life; now it's time to share it with the world!

By verifying your smart contract on Etherscan, anyone can view your source code and interact with your smart contract. Let's get started!

Step 1: Generate an API Key on your Etherscan account {#step-1-generate-an-api-key-on-your-etherscan-account}

An Etherscan API Key is necessary to verify that you own the smart contract you are trying to publish.

If you don't have an Etherscan account already, sign up for an account.

Once logged in, find your username in the navigation bar, hover over it and select the **My profile** button.

On your profile page, you should see a side navigation bar. From the side navigation bar, select **API Keys**. Next, press the "Add" button to create a new API key, name your app **hello-world** and press the **Create New API Key** button.

Your new API key should appear in the API key table. Copy the API key to your clipboard.

Next, we need to add the Etherscan API key to our `.env` file.

After adding it, your `.env` file should look like this:

```
javascript API_URL = "https://eth-goerli.alchemyapi.io/v2/your-api-key" PUBLIC_KEY = "your-public-account-address"
PRIVATE_KEY = "your-private-account-address" CONTRACT_ADDRESS = "your-contract-address" ETHERSCAN_API_KEY = "your-
etherscan-key"
```

Hardhat-deployed smart contracts {#hardhat-deployed-smart-contracts}

Install hardhat-etherscan {#install-hardhat-etherscan}

Publishing your contract to Etherscan using Hardhat is straightforward. You will first need to install the `hardhat-etherscan` plugin to get started. `hardhat-etherscan` will automatically verify the smart contract's source code and ABI on Etherscan. To add this, in the `hello-world` directory run:

```
text npm install --save-dev @nomiclabs/hardhat-etherscan
```

Once installed, include the following statement at the top of your `hardhat.config.js`, and add the Etherscan config options:

```javascript // hardhat.config.js

require("dotenv").config() require("@nomiclabs/hardhat-ethers") require("@nomiclabs/hardhat-etherscan")

const { API_URL, PRIVATE_KEY, ETHERSCAN_API_KEY } = process.env

module.exports = { solidity: "0.7.3", defaultNetwork: "goerli", networks: { hardhat: {}, goerli: { url: API_URL, accounts: [`0x${PRIVATE_KEY}`], }, }, etherscan: { // Your API key for Etherscan // Obtain one at https://etherscan.io/ apiKey: ETHERSCAN_API_KEY, }, } ```

### Verify your smart contract on Etherscan {#verify-your-smart-contract-on-etherscan}

Ensure all files are saved and all `.env` variables are correctly configured.

Run the `verify` task, passing the contract address, and the network to where it's deployed:

```
text npx hardhat verify --network goerli DEPLOYED_CONTRACT_ADDRESS 'Hello World!'
~~~

Make sure that `DEPLOYED_CONTRACT_ADDRESS` is the address of your deployed smart contract on the Goerli test network. Also, the final argument (`'Hello World!'`) must be the same string value used during the deploy step in part 1.

If all goes well, you will see the following message in your terminal:

```text Successfully submitted source code for contract contracts/HelloWorld.sol:HelloWorld at 0xdeployed-contract-address for verification on Etherscan. Waiting for verification result...

Successfully verified contract HelloWorld on Etherscan. https://goerli.etherscan.io/address#contracts ```

Congrats! Your smart contract code is on Etherescan!
~~~

### Check out your smart contract on Etherscan! {#check-out-your-smart-contract-on-etherscan}

When you navigate to the link provided in your terminal, you should be able to see your smart contract code and ABI published on Etherscan!

**Wahooo - you did it champ! Now anyone can call or write to your smart contract! We can't wait to see what you build next!**

## Part 4 - Integrating your smart contract with the frontend {#part-4-integrating-your-smart-contract-with-the-frontend}

By the end of this tutorial, you'll know how to:

- Connect a MetaMask wallet to your dapp
- Read data from your smart contract using the Alchemy Web3 API
- Sign Ethereum transactions using MetaMask

For this dapp, we'll be using React as our frontend framework; however, it's important to note that we won't be spending much time breaking down its fundamentals, as we'll mostly be focusing on bringing Web3 functionality to our project.

As a prerequisite, you should have a beginner-level understanding of React. If not, we recommend completing the official Intro to React tutorial.

### Clone the starter files {#clone-the-starter-files}

First, go to the hello-world-part-four GitHub repository to get the starter files for this project and clone this repository to your local machine.

Open the cloned repository locally. Notice that it contains two folders: `starter-files` and `completed`.

- `starter-files` - **we will be working in this directory**, we will connect the UI to your Ethereum wallet and the smart contract we published to Etherscan in Part 3.
- `completed` contains the entire completed tutorial and should only be used as a reference if you get stuck.

Next, open your copy of `starter-files` to your favorite code editor, and then navigate into the `src` folder.

All of the code we'll write will live under the `src` folder. We'll be editing the `HelloWorld.js` component and the `util/interact.js` JavaScript files to give our project Web3 functionality.

### Check out the starter files {#check-out-the-starter-files}

Before we start coding, let's explore what is provided to us in the starter files.

#### Get your react project running {#get-your-react-project-running}

Let's start by running the React project in our browser. The beauty of React is that once we have our project running in our browser, any changes we save will be updated live in our browser.

To get the project running, navigate to the root directory of the `starter-files` folder, and the run `npm install` in your terminal to install the dependencies of the project:

```
bash cd starter-files npm install
```

Once those have finished installing, run `npm start` in your terminal:

```
bash npm start
```

Doing so should open http://localhost:3000/ in your browser, where you'll see the frontend for our project. It should consist of one field (a place to update the message stored in your smart contract), a "Connect Wallet" button, and an "Update" button.

If you try clicking either button, you'll notice that they don't work—that's because we still need to program their functionality.

The `HelloWorld.js` component {#the-helloworld-js-component}

Let's go back into the `src` folder in our editor and open the `HelloWorld.js` file. It's super important that we understand everything in this file, as it is the primary React component we will be working on.

At the top of this file, you'll notice we have several import statements that are necessary to get our project running, including the React library, useEffect and useState hooks, some items from the `./util/interact.js` (we'll describe them in more details soon!), and the Alchemy logo.

```javascript // HelloWorld.js

import React from "react" import { useEffect, useState } from "react" import { helloWorldContract, connectWallet, updateMessage, loadCurrentMessage, getCurrentWalletConnected, } from "./util/interact.js"

import alchemylogo from "./alchemylogo.svg" ```

Next, we have our state variables that we will update after specific events.

```javascript // HelloWorld.js

//State variables const [walletAddress, setWallet] = useState("") const [status, setStatus] = useState("") const [message, setMessage] = useState("No connection to the network.") const [newMessage, setNewMessage] = useState("") ```

Here's what each of the variables represents:

- `walletAddress` - a string that stores the user's wallet address
- `status`- a string that stores a helpful message that guides the user on how to interact with the dapp
- `message` - a string that stores the current message in the smart contract
- `newMessage` - a string that stores the new message that will be written to the smart contract

After the state variables, you'll see five un-implemented functions:`useEffect` ,`addSmartContractListener`, `addWalletListener` , `connectWalletPressed`, and `onUpdatePressed`. We'll explain what they do below:

```javascript // HelloWorld.js

//called only once useEffect(async () => { //TODO: implement }, [])

function addSmartContractListener() { //TODO: implement }

function addWalletListener() { //TODO: implement }

const connectWalletPressed = async () => { //TODO: implement }

const onUpdatePressed = async () => { //TODO: implement } ```

- `useEffect`- this is a React hook that is called after your component is rendered. Because it has an empty array`[]` prop passed into it (see line 4), it will only be called on the component's *first* render. Here we'll load the current message stored in our smart contract, call our smart contract and wallet listeners, and update our UI to reflect whether a wallet is already connected.
- `addSmartContractListener`- this function sets up a listener that will watch for our HelloWorld contract's`UpdatedMessages` event and update our UI when the message is changed in our smart contract.
- `addWalletListener`- this function sets up a listener that detects changes in the user's MetaMask wallet state, such as when the user disconnects their wallet or switches addresses.
- `connectWalletPressed`- this function will be called to connect the user's MetaMask wallet to our dapp.
- `onUpdatePressed` - this function will be called when the user wants to update the message stored in the smart contract.

Near the end of this file, we have the UI of our component.

```javascript // HelloWorld.js

//the UI of our component return (

{walletAddress.length > 0 ? ( "Connected: " + String(walletAddress).substring(0, 6) + "..." + String(walletAddress).substring(38) ) : ( Connect Wallet )}

## Current Message:

{message}

## New Message:

Update the message in yo setNewMessage(e.target.value)} value={newMessage} />

{status}

Update

) ```

If you scan this code carefully, you'll notice where we use our various state variables in our UI:

- On lines 6-12, if the user's wallet is connected (i.e.`walletAddress.length > 0`), we display a truncated version of the user `walletAddress` in the button with ID "walletButton;" otherwise it simply says "Connect Wallet."
- On line 17, we display the current message stored in the smart contract, which is captured in the`message` string.
- On lines 23-26, we use a[controlled component] to update our `newMessage` state variable when the input in the text field changes.

In addition to our state variables, you'll also see that`connectWalletPressed` and `onUpdatePressed` functions are called when the buttons with IDs `publishButton` and `walletButton` are clicked respectively.

Finally, let's address where is this `HelloWorld.js` component added.

If you go to the `App.js` file, which is the main component in React that acts as a container for all other components, you'll see that our `HelloWorld.js` component is injected on line 7.

Last but not least, let's check out one more file provided for you, the`interact.js` file.

#### The `interact.js` file {#the-interact-js-file}

Because we want to prescribe to the[M-V-C] paradigm, we'll want a separate file that contains all our functions to manage the logic, data, and rules of our dapp, and then be able to export those functions to our frontend (our `HelloWorld.js` component).

This is the exact purpose of our `interact.js` file!

Navigate to the `util` folder in your `src` directory, and you'll notice we've included a file called `interact.js` that will contain all of our smart contract interaction and wallet functions and variables.

```javascript // interact.js

//export const helloWorldContract;

export const loadCurrentMessage = async () => {}

export const connectWallet = async () => {}

const getCurrentWalletConnected = async () => {}

export const updateMessage = async (message) => {} ```

You'll notice at the top of the file that we've commented out the`helloWorldContract` object. Later in this tutorial, we will uncomment this object and instantiate our smart contract in this variable, which we will then export into our `HelloWorld.js` component.

The four unimplemented functions after our `helloWorldContract` object do the following:

- `loadCurrentMessage` - this function handles the logic of loading the current message stored in the smart contract. It will make a *read* call to the Hello World smart contract using the[Alchemy Web3 API].
- `connectWallet` - this function will connect the user's MetaMask to our dapp.
```

- `getCurrentWalletConnected` - this function will check if an Ethereum account is already connected to our dapp on page load and update our UI accordingly.
- `updateMessage` - this function will update the message stored in the smart contract. It will make a *write* call to the Hello World smart contract, so the user's MetaMask wallet will have to sign an Ethereum transaction to update the message.

Now that we understand what we're working with, let's figure out how to read from our smart contract!

Step 3: Read from your smart contract {#step-3-read-from-your-smart-contract}

To read from your smart contract, you'll need to successfully set up:

- An API connection to the Ethereum chain
- A loaded instance of your smart contract
- A function to call to your smart contract function
- A listener to watch for updates when the data you're reading from the smart contract changes

This may sounds like a lot of steps, but don't worry! We'll walk you through how to do each of them step-by-step! :)

Establish an API connection to the Ethereum chain {#establish-an-api-connection-to-the-ethereum-chain}

So remember how in Part 2 of this tutorial, we used our Alchemy Web3 key to read from our smart contract? You'll also need an Alchemy Web3 key in your dapp to read from the chain.

If you don't have it already, first install Alchemy Web3 by navigating to the root directory of your `starter-files` and running the following in your terminal:

`text npm install @alch/alchemy-web3`

Alchemy Web3 is a wrapper around Web3.js, providing enhanced API methods and other crucial benefits to make your life as a web3 developer easier. It is designed to require minimal configuration so you can start using it in your app right away!

Then, install the dotenv package in your project directory, so we have a secure place to store our API key after we fetch it.

`text npm install dotenv --save`

For our dapp, **we'll be using our Websockets API key** instead of our HTTP API key, as it will allow us to set up a listener that detects when the message stored in the smart contract changes.

Once you have your API key, create a `.env` file in your root directory and add your Alchemy Websockets url to it. Afterwards, your `.env` file should look like so:

`javascript REACT_APP_ALCHEMY_KEY = wss://eth-goerli.ws.alchemyapi.io/v2/<key>`

Now, we're ready to set up our Alchemy Web3 endpoint in our dapp! Let's go back to our `interact.js`, which is nested inside our `util` folder and add the following code at the top of the file:

```javascript // interact.js

require("dotenv").config() const alchemyKey = process.env.REACT_APP_ALCHEMY_KEY const { createAlchemyWeb3 } = require("@alch/alchemy-web3") const web3 = createAlchemyWeb3(alchemyKey)

//export const helloWorldContract; ```

Above, we first imported the Alchemy key from our `.env` file and then passed our `alchemyKey` to `createAlchemyWeb3` to establish our Alchemy Web3 endpoint.

With this endpoint ready, it's time to load our smart contract!

### Loading your Hello World smart contract {#loading-your-hello-world-smart-contract}

To load your Hello World smart contract, you'll need its contract address and ABI, both of which can be found on Etherscan if you completed Part 3 of this tutorial.

### How to get your contract ABI from Etherscan {#how-to-get-your-contract-abi-from-etherscan}
```

If you skipped Part 3 of this tutorial, you can use the HelloWorld contract with address 0x6f3f635A9762B47954229Ea479b4541eAF402A6A. It's ABI can be found here.

A contract ABI is necessary for specifying which function a contract will invoke as well ensuring that the function will return data in the format you're expecting. Once we've copied our contract ABI, let's save it as a JSON file called `contract-abi.json` in your `src` directory.

Your contract-abi.json should be stored in your src folder.

Armed with our contract address, ABI, and Alchemy Web3 endpoint, we can use the contract method to load an instance of our smart contract. Import your contract ABI into the `interact.js` file and add your contract address.

```javascript // interact.js

const contractABI = require("../contract-abi.json") const contractAddress = "0x6f3f635A9762B47954229Ea479b4541eAF402A6A" ```

We can now finally uncomment our `helloWorldContract` variable, and load the smart contract using our AlchemyWeb3 endpoint:

`javascript // interact.js export const helloWorldContract = new web3.eth.Contract( contractABI, contractAddress )`

To recap, the first 12 lines of your `interact.js` should now look like this:

```javascript // interact.js

require("dotenv").config() const alchemyKey = process.env.REACT_APP_ALCHEMY_KEY const { createAlchemyWeb3 } = require("@alch/alchemy-web3") const web3 = createAlchemyWeb3(alchemyKey)

const contractABI = require("../contract-abi.json") const contractAddress = "0x6f3f635A9762B47954229Ea479b4541eAF402A6A"

export const helloWorldContract = new web3.eth.Contract( contractABI, contractAddress ) ```

Now that we have our contract loaded, we can implement our `loadCurrentMessage` function!

**Implementing `loadCurrentMessage` in your `interact.js` file {#implementing-loadCurrentMessage-in-your-interact-js-file}**

This function is super simple. We're going make a simple async web3 call to read from our contract. Our function will return the message stored in the smart contract:

Update the `loadCurrentMessage` in your `interact.js` file to the following:

```javascript // interact.js

export const loadCurrentMessage = async () => { const message = await helloWorldContract.methods.message().call() return message } ```

Since we want to display this smart contract in our UI, let's update the `useEffect` function in our `HelloWorld.js` component to the following:

```javascript // HelloWorld.js

//called only once useEffect(async () => { const message = await loadCurrentMessage() setMessage(message) }, []) ```

Note, we only want our `loadCurrentMessage` to be called once during the component's first render. We'll soon implement `addSmartContractListener` to automatically update the UI after the message in the smart contract changes.

Before we dive into our listener, let's check out what we have so far! Save your `HelloWorld.js` and `interact.js` files, and then go to http://localhost:3000/

You'll notice that the current message no longer says "No connection to the network." Instead it reflects the message stored in the smart contract. Sick!

**Your UI should now reflect the message stored in the smart contract {#your-UI-should-now-reflect-the-message-stored-in-the-smart-contract}**

Now speaking of that listener...

**Implement `addSmartContractListener` {#implement-addsmartcontractlistener}**

If you think back to the `HelloWorld.sol` file we wrote in [Part 1 of this tutorial series](), you'll recall that there is a smart contract event called `UpdatedMessages` that is emitted after our smart contract's `update` function is invoked (see lines 9 and 27):

```javascript // HelloWorld.sol

// Specifies the version of Solidity, using semantic versioning. // Learn more: https://solidity.readthedocs.io/en/v0.5.10/layout-of-source-files.html#pragma pragma solidity ^0.7.3;

// Defines a contract named `HelloWorld`. // A contract is a collection of functions and data (its state). Once deployed, a contract resides at a specific address on the Ethereum blockchain. Learn more: https://solidity.readthedocs.io/en/v0.5.10/structure-of-a-contract.html contract HelloWorld {

//Emitted when update function is called //Smart contract events are a way for your contract to communicate that something happened on the blockchain to your app front-end, which can be 'listening' for certain events and take action when they happen. event UpdatedMessages(string oldStr, string newStr);

// Declares a state variable `message` of type `string`. // State variables are variables whose values are permanently stored in contract storage. The keyword `public` makes variables accessible from outside a contract and creates a function that other contracts or clients can call to access the value. string public message;

// Similar to many class-based object-oriented languages, a constructor is a special function that is only executed upon contract creation. // Constructors are used to initialize the contract's data. Learn more:https://solidity.readthedocs.io/en/v0.5.10/contracts.html#constructors constructor(string memory initMessage) {

```
  // Accepts a string argument `initMessage` and sets the value into the contract's `message` storage variable).
  message = initMessage;
~~~

}

// A public function that accepts a string argument and updates the `message` storage variable. function update(string memory newMessage) public { string memory oldMsg = message; message = newMessage; emit UpdatedMessages(oldMsg, newMessage); } } ```

Smart contract events are a way for your contract to communicate that something happened (i.e. there was an *event*) on the blockchain to your front-end application, which can be 'listening' for specific events and take action when they happen.

The `addSmartContractListener` function is going to specifically listen for our Hello World smart contract's `UpdatedMessages` event, and update our UI to display the new message.

Modify `addSmartContractListener` to the following:

```javascript // HelloWorld.js

function addSmartContractListener() { helloWorldContract.events.UpdatedMessages({}, (error, data) => { if (error) { setStatus("😥 " + error.message) } else { setMessage(data.returnValues[1]) setNewMessage("") setStatus("                    Your message has been updated!") } }) } ```

Let's break down what happens when the listener detects an event:

- If an error occurs when the event is emitted, it will be reflected in the UI via our `status` state variable.
- Otherwise, we will use the `data` object returned. The `data.returnValues` is an array indexed at zero where the first element in the array stores the previous message and second element stores the updated one. Altogether, on a successful event we'll set our `message` string to the updated message, clear the `newMessage` string, and update our `status` state variable to reflect that a new message has been published on our smart contract.

Finally, let's call our listener in our `useEffect` function so it is initialized on the `HelloWorld.js` component's first render. Altogether, your `useEffect` function should look like this:

```javascript // HelloWorld.js
~~~

useEffect(async () => { const message = await loadCurrentMessage() setMessage(message) addSmartContractListener() }, []) ```

Now that we're able to read from our smart contract, it would be great to figure out how to write to it too! However, to write to our dapp, we must first have an Ethereum wallet connected to it.

So, next we'll tackle setting up our Ethereum wallet (MetaMask) and then connecting it to our dapp!

## Step 4: Set up your Ethereum wallet {#step-4-set-up-your-ethereum-wallet}

To write anything to the Ethereum chain, users must sign transactions using their virtual wallet's private keys. For this tutorial, we'll use MetaMask, a virtual wallet in the browser used to manage your Ethereum account address, as it makes this transaction signing super easy for the end-user.

If you want to understand more about how transactions on Ethereum work, check outthis page from the Ethereum foundation.

### Download MetaMask {#download-metamask}

You can download and create a MetaMask account for freehere. When you are creating an account, or if you already have an account, make sure to switch over to the “Goerli Test Network” in the upper right (so that we're not dealing with real money).

### Add ether from a Faucet {#add-ether-from-a-faucet}

To sign a transaction on the Ethereum blockchain, we'll need some fake Eth. To get Eth you can go to theFaucETH and enter your Goerli account address, click “Request funds”, then select “Ethereum Testnet Goerli” in the dropdown and finally click “Request funds” button again. You should see Eth in your MetaMask account soon after!

### Check your Balance {#check-your-balance}

To double check our balance is there, let's make aneth_getBalance request using Alchemy's composer tool. This will return the amount of Eth in our wallet. After you input your MetaMask account address and click “Send Request”, you should see a response like this:

```
text {"jsonrpc": "2.0", "id": 0, "result": "0xde0b6b3a7640000"}
```

NOTE: This result is in wei not eth. Wei is used as the smallest denomination of ether. The conversion from wei to eth is: 1 eth = 10^{18} wei. So if we convert 0xde0b6b3a7640000 to decimal we get $1*10^{18}$ which equals 1 eth.

Phew! Our fake money is all there!

Step 5: Connect MetaMask to your UI {#step-5-connect-metamask-to-your-UI}

Now that our MetaMask wallet is set up, let's connect our dapp to it!

The `connectWallet` function {#the-connectWallet-function}

In our `interact.js`file, let's implement the `connectWallet` function, which we can then call in our`HelloWorld.js` component.

Let's modify `connectWallet` to the following:

```javascript // interact.js

export const connectWallet = async () => { if (window.ethereum) { try { const addressArray = await window.ethereum.request({ method: "eth_requestAccounts", }) const obj = { status: " Write a message in the text-field above.", address: addressArray[0], } return obj } catch (err) { return { address: "", status: "😥 " + err.message, } } } else { return { address: "", status: (

{" "} https://metamask.io/download.html}> You must install MetaMask, a virtual Ethereum wallet, in your browser.

), } } } ```

So what does this giant block of code do exactly?

Well, first, it checks if it `window.ethereum` is enabled in your browser.
```

`window.ethereum` is a global API injected by MetaMask and other wallet providers that allows websites to request users' Ethereum accounts. If approved, it can read data from the blockchains the user is connected to, and suggest that the user sign messages and transactions . Check out the [MetaMask docs] for more info!

If `window.ethereum` *is not* present, then that means MetaMask is not installed. This results in a JSON object being returned, where `address` returned is an empty string, and the `status` JSX object relays that the user must install MetaMask.

Now if `window.ethereum` *is* present, then that's when things get interesting.

Using a try/catch loop, we'll try to connect to MetaMask by calling `window.ethereum.request({ method: "eth_requestAccounts" });`. Calling this function will open up MetaMask in the browser, whereby the user will be prompted to connect their wallet to your dapp.

- If the user chooses to connect, `method: "eth_requestAccounts"` will return an array that contains all of the user's account addresses that connected to the dapp. Altogether, our `connectWallet` function will return a JSON object that contains the *first* `address` in this array (see line 9) and a `status` message that prompts the user to write a message to the smart contract.
- If the user rejects the connection, then the JSON object will contain an empty string for the `address` returned and a `status` message that reflects that the user rejected the connection.

Now that we've written this `connectWallet` function, the next step is to call it to our `HelloWorld.js` component.

Add the `connectWallet` function to your `HelloWorld.js` UI Component {#add-the-connectWallet-function-to-your-HelloWorld-js-ui-component}

Navigate to the `connectWalletPressed` function in `HelloWorld.js`, and update it to the following:

```javascript // HelloWorld.js

const connectWalletPressed = async () => { const walletResponse = await connectWallet() setStatus(walletResponse.status) setWallet(walletResponse.address) } ```

Notice how most of our functionality is abstracted away from our `HelloWorld.js` component from the `interact.js` file? This is so we comply with the M-V-C paradigm!

In `connectWalletPressed`, we simply make an await call to our imported `connectWallet` function, and using its response, we update our `status` and `walletAddress` variables via their state hooks.

Now, let's save both files (`HelloWorld.js` and `interact.js`) and test out our UI so far.

Open your browser on the [http://localhost:3000/] page, and press the "Connect Wallet" button on the top right of the page.

If you have MetaMask installed, you should be prompted to connect your wallet to your dapp. Accept the invitation to connect.

You should see that the wallet button now reflects that your address is connected! Yasssss

Next, try refreshing the page... this is strange. Our wallet button is prompting us to connect MetaMask, even though it is already connected...

However, have no fear! We easily can address that (get it?) by implementing `getCurrentWalletConnected`, which will check if an address is already connected to our dapp and update our UI accordingly!

**The `getCurrentWalletConnected` function {#the-getcurrentwalletconnected-function}**

Update your `getCurrentWalletConnected` function in the `interact.js` file to the following:

```javascript // interact.js

export const getCurrentWalletConnected = async () => { if (window.ethereum) { try { const addressArray = await window.ethereum.request({ method: "eth_accounts", }) if (addressArray.length > 0) { return { address: addressArray[0], status: "                                                    Write a message in the text-field above.", } } else { return { address: "", status: "                         Connect to MetaMask using the top right button.", } } } catch (err) { return { address: "", status: "😥 " + err.message, } } } else { return { address: "", status: (

{" "}                                    [https://metamask.io/download.html}> You must install MetaMask, a virtual Ethereum wallet, in your browser.]

), } } } ```

This code is *very* similar to the `connectWallet` function we just wrote in the previous step.

The main difference is that instead of calling the method `eth_requestAccounts`, which opens MetaMask for the user to connect their wallet, here we call the method `eth_accounts`, which simply returns an array containing the MetaMask addresses currently connected to our dapp.

To see this function in action, let's call it in our `useEffect` function of our `HelloWorld.js` component:

```javascript // HelloWorld.js

useEffect(async () => { const message = await loadCurrentMessage() setMessage(message) addSmartContractListener()

const { address, status } = await getCurrentWalletConnected() setWallet(address) setStatus(status) }, []) ```

Notice, we use the response of our call to `getCurrentWalletConnected` to update our `walletAddress` and `status` state variables.

Now that you've added this code, let's try refreshing our browser window.

Niceeeee! The button should say that you're connected, and show a preview of your connected wallet's address - even after you refresh!

**Implement `addWalletListener` {#implement-addwalletlistener}**

The final step in our dapp wallet setup is implementing the wallet listener so our UI updates when our wallet's state changes, such as when the user disconnects or switches accounts.

In your `HelloWorld.js` file, modify your `addWalletListener` function as the following:

```javascript // HelloWorld.js

function addWalletListener() { if (window.ethereum) { window.ethereum.on("accountsChanged", (accounts) => { if (accounts.length > 0) { setWallet(accounts[0]) setStatus(" Write a message in the text-field above.") } else { setWallet("") setStatus(" Connect to MetaMask using the top right button.") } }) } else { setStatus(

{" "} https://metamask.io/download.html}> You must install MetaMask, a virtual Ethereum wallet, in your browser.

) } } ```

I bet you don't even need our help to understand what's going on here at this point, but for thoroughness purposes, let's quickly break it down:

- First, our function checks if `window.ethereum` is enabled (i.e. MetaMask is installed).
- If it's not, we simply set our `status` state variable to a JSX string that prompts the user to install MetaMask.
- If it is enabled, we set up the listener `window.ethereum.on("accountsChanged")` on line 3 that listens for state changes in the MetaMask wallet, which include when the user connects an additional account to the dapp, switches accounts, or disconnects an account. If there is at least one account connected, the `walletAddress` state variable is updated as the first account in the `accounts` array returned by the listener. Otherwise, `walletAddress` is set as an empty string.

Last but not least, we must call it in our `useEffect` function:

```javascript // HelloWorld.js

useEffect(async () => { const message = await loadCurrentMessage() setMessage(message) addSmartContractListener()

const { address, status } = await getCurrentWalletConnected() setWallet(address) setStatus(status)

addWalletListener() }, []) ```

And that's it! We've successfully completed programming all of our wallet functionality! Now onto our last task: updating the message stored in our smart contract!

## Step 6: Implement the `updateMessage` function {#step-6-implement-the-updateMessage-function}
```

Alrighty fam, we've arrived at the home stretch! In the`updateMessage` of your `interact.js` file, we're going to do the following:

1. Make sure the message we wish to publish in our smart contact is valid
2. Sign our transaction using MetaMask
3. Call this function from our `HelloWorld.js` frontend component

This won't take very long; let's finish this dapp!

Input error handling {#input-error-handling}

Naturally, it makes sense to have some sort of input error handling at the start of the function.

We'll want our function to return early if there is no MetaMask extension installed, there is no wallet connected (i.e. the`address` passed in is an empty string), or the `message` is an empty string. Let's add the following error handling to`updateMessage`:

```javascript // interact.js

export const updateMessage = async (address, message) => { if (!window.ethereum || address === null) { return { status: "                              Connect your MetaMask wallet to update the message on the blockchain.", } }

if (message.trim() === "") { return { status: "❌ Your message cannot be an empty string.", } } } ```

Now that it have proper input error handling, it's time to sign the transaction via MetaMask!

**Signing our transaction {#signing-our-transaction}**

If you're already comfortable with traditional web3 Ethereum transactions, the code we write next will be very familiar. Below your input error handling code, add the following to `updateMessage`:

```javascript // interact.js

//set up transaction parameters const transactionParameters = { to: contractAddress, // Required except during contract publications. from: address, // must match user's active address. data: helloWorldContract.methods.update(message).encodeABI(), }

//sign the transaction try { const txHash = await window.ethereum.request({ method: "eth_sendTransaction", params: [transactionParameters], }) return { status: ( ✅{" "} https://goerli.etherscan.io/tx/${txHash}}> View the status of your transaction on Etherscan!
ℹ️ Once the transaction is verified by the network, the message will be updated automatically. ), } } catch (error) { return { status: "😥 " + error.message, } } ```

Let's breakdown what's happening. First, we set up our transactions parameters, where:

- `to` specifies the recipient address (our smart contract)
- `from` specifies the signer of the transaction, the`address` variable we passed into our function
- `data` contains the call to our Hello World smart contract's`update` method, receiving our `message` string variable as input

Then, we make an await call,`window.ethereum.request`, where we ask MetaMask to sign the transaction. Notice, on lines 11 and 12, we're specifying our eth method, `eth_sendTransaction`and passing in our `transactionParameters`.

At this point, MetaMask will open up in the browser, and prompt the user to sign or reject the transaction.

- If the transaction is successful, the function will return a JSON object where the`status` JSX string prompts the user to check out Etherscan for more information about their transaction.
- If the transaction fails, the function will return a JSON object where the`status` string relays the error message.

Altogether, our `updateMessage` function should look like this:

```javascript // interact.js

export const updateMessage = async (address, message) => { //input error handling if (!window.ethereum || address === null) { return { status: "                              Connect your MetaMask wallet to update the message on the blockchain.", } }

if (message.trim() === "") { return { status: "❌ Your message cannot be an empty string.", } }
```

//set up transaction parameters const transactionParameters = { to: contractAddress, // Required except during contract publications. from: address, // must match user's active address. data: helloWorldContract.methods.update(message).encodeABI(), }

//sign the transaction try { const txHash = await window.ethereum.request({ method: "eth_sendTransaction", params: [transactionParameters], }) return { status: (✅{" "} https://goerli.etherscan.io/tx/${txHash}}> View the status of your transaction on Etherscan!
ℹ️ Once the transaction is verified by the network, the message will be updated automatically.), } } catch (error) { return { status: "😥 " + error.message, } } } ```

Last but not least, we need to connect our `updateMessage` function to our `HelloWorld.js` component.

Connect `updateMessage` to the `HelloWorld.js` frontend {#connect-updatemessage-to-the-helloworld-js-frontend}

Our `onUpdatePressed` function should make an await call to the imported `updateMessage` function and modify the `status` state variable to reflect whether our transaction succeeded or failed:

```javascript // HelloWorld.js

const onUpdatePressed = async () => { const { status } = await updateMessage(walletAddress, newMessage) setStatus(status) } ```

It's super clean and simple. And guess what... YOUR DAPP IS COMPLETE!!!

Go ahead and test out the **Update** button!

## Make your own custom dapp {#make-your-own-custom-dapp}

Wooooo, you made it to the end of the tutorial! To recap, you learned how to:

- Connect a MetaMask wallet to your dapp project
- Read data from your smart contract using the Alchemy Web3 API
- Sign Ethereum transactions using MetaMask

Now you're fully equipped to apply the skills from this tutorial to build out your own custom dapp project! As always, if you have any questions, don't hesitate to reach out to us for help in the Alchemy Discord. ♂

Once you complete this tutorial, let us know how your experience was or if you have any feedback by tagging us on Twitter @alchemyplatform!
```