Blockchain Developer

Table of Contents

7 Openzeppelin

They are a company providing libraries to **standardize smart contracts**, like in the first years people used to standardize protocols. It's the only way to go for a more secure world, and less hacks. All contracts are published on github and can be seen by everyone, and are audited periodically. They make money by auditing the code produced by other companies.

8 Metamask

It's a common wallet used to manage crypto accounts and transfers, even though it's not that secure. I wouldn't use it to store real money, but it's necessary to transfer some test money and deploy smart contracts on testnets. It's often used as a 'warm wallet' containing small amounts of money, that can potentially be hacked without loosing your entire life's savings. 'Cold wallets' are for most of the time disconnected from the Internet, and used to transfer money from/to exchanges or other wallets in general.
Another more secure type of wallet is the 'multi signature' wallet, in this case you need to provide multiple private keys to perform a transaction, of course these keys should be stored in different places and kep secure, and not be lost. Brute forcing private keys is considered hard if not feasible at all, that's the security behind cryptocurrencies in general and Bitcoin too. Some examples of **multi-sig wallets** are the following:

1. Armory
2. BitGo
3. Coinbase
4. CoPay
5. Electrum
6. **Gnosis Safe**

Ledger and Trezor are **cold wallets** instead, making your life (intentionally) much more difficult to transfer money.

8.1 MetaMask: a different model of account security

Public blockchain technology uses a very different set of tools to secure user accounts, compared to traditional online technologies. Most of us are used to creating an account with an app, or service, or what have you, and being able to, for example, write to Customer Support to reset our password, or username; we're used to the app keeping our data, presumably on some sort of computer that belongs to the company. Well... MetaMask **doesn't work like that**. MetaMask has three different types of secret that are used in different ways to keep your wallet, and your accounts, private and safe: **The Secret Recovery Phrase**, the password, and private keys. We'll walk you through these secrets one at a time.

8.1.1 Intro to Secret Recovery Phrases

One of the key (you'll see what I did there) technologies underlying MetaMask, and in fact, most user account-related tools in the crypto space is that of a *seed phrase*, or as it's referred to in MetaMask, your ***Secret Recovery Phrase***.
First, the technical explanation: Seed phrases as we know them today were codified for usage in Bitcoin, according to a standard referred to as Bitcoin Improvement Proposal 39, or BIP-39. In simple terms, a series of words are selected with a high level of randomness from a specific list of words. In MetaMask and many other Ethereum-compatible technologies, there are 12 words in a seed phrase. Some older seeds generated by the Brave browser, and some hardware wallets, use 24-word phrases. Each one of these words corresponds to a series of numbers, and when placed in **a specific order**, represent a much more user-friendly way to remember a very, very long number. That number is the private key to your accounts. (...now you see what I did there?).

8.1.2 There are a number of important features to note here:

- The **Secret Recovery Phrase is the key to the wallet**. If someone has the key, they have complete access to the wallet. **MetaMask does not keep the keys: you are the custodian of your wallet.** MetaMask will **never** ask for your Secret Recovery Phrase, even in a customer support scenario. If someone does ask for it, they are likely trying to scam you or steal your funds.
- Your secret recovery phrase is **used locally to derive private keys**, one per account/address. **Accounts are stored on the blockchain, and these private keys unlock those accounts.**
- **If you uninstall the app**, or the extension, then the local version of the data is gone (the notable exception being the vault), but any transactions you performed with that local version of MetaMask will have been recorded on the blockchain. Therefore, the transactions should be reflected both on a block explorer, and in another instance of MetaMask, so long as you restore using the same Secret Recovery Phrase (**with the words in the same order**). This means that so long as you have your Secret Recovery Phrase, you will always be able to uninstall MetaMask and restore your wallet.
- **Within your wallet, you can have a very large number of separate accounts.** When MetaMask creates or restores your wallet from the Secret Recovery Phrase, it initially produces only the first account. However, any additional accounts you create can be re-created in a future instance of MetaMask; **as the wallet is *deterministic*, it will always re-create the same accounts, in the same order. For more on this issue, see the FAQs below.**
- **It is possible to import accounts from other Ethereum-compatible technologies into a MetaMask wallet.** In order to do so, the *private key* of that specific account is used. However, **this account will not be automatically restored by MetaMask in another instance; you will have to manually re-add it**. Therefore, if you have manually imported accounts, **make note of their private keys, in the same way you did your seed phrase**, in order to be able to re-import them in the future.

8.1.3 MetaMask Secret Recovery Phrase: DOs and DON'Ts

DO	DON'T
Write down your Secret Recovery Phrase somewhere safe	Keep it in an easily discovered location; e.g. in a cloud-saved document or email titled "Seed Phrase"; on a post-it note stuck to your computer
Double-check your spelling and that you wrote down every word in the same order they were given	Change the order of the words
Reach out to MetaMask for customer assistance when needed	Provide your seed phrase to anyone, even if they say they're from customer service

9 Remix

It has a local virtual machine to test smart contracts, can connect to other external blockchains, installing an extension could save files outside those of the browser. It’s something you can play with, but not for a professional use. Moreover, many operations are still manual and you would waste a lot of time changing and debugging a smart contract, re-compiling it, re-deploying it and so on. Anyway, for people who want to play with it, just to start writing simple smart contracts, it’s the best tool because it’s already there and you don’t have to install anything.

10 Frontend interfaces

You can have and use many different languages to build your UI/UX, even though the most commons are for sure Javascript and Typescript (same syntax of Javascript, but with strong Typing). **React** is a Javascript library that was created by Facebook, and is focused on providing graphical User Interfaces. As we have already

explained in the beginning of this document, there are people who build their entire career as 'frontend developers' because it's a wide area that requires experience of **months or years** of programming. Beware about it, if you'll ever need to hire someone, or put up a whole development group: don't search for white unicorn, they don't exist. And those few, want to be paid. A LOT. They are usually **monks** that do not have Youtube channels, do not pass the whole day posting stuff on LinkedIn, tweeting, instagramming their unbelievable achievements.
Given the fact that most frontend developers use Javascript or similar languages and libraries (Typescript, React), some tools to interact with blockchains became popular too:

- web3.js
- ether.js

... and will be covered later in this chapter (the first one in more detail). This guide doesn't want to be a comprehensive one (we would need many books to really cover everything), but hopefully will provide useful resources and examples to have a 360 degrees idea of the whole process and tech stuff involved in building a Dapp (web3.0 decentralized application).
Let's be honest: if you are interested by technologies, there is not much to learn here. Far away to say the UI/UX is not fundamental on every internet site nowadays, you'll loose your customers if you don't put enough efforts on it.
But the short long story is that you will import already defined 'containers', and describe their aspect, depending on events that happen on the screen (for example 'Mouse over', 'double click', ...). Every event can trigger a certain action and launch, for example, the Metamask external application.

If you use React with **Tailwind CSS**, your pages will look something like this:

```
import React, { useState } from "react";
import { Transition } from "@headlessui/react";

function Nav() {
  const [isOpen, setIsOpen] = useState(false);
  return (
    <div>
      <nav className="bg-gray-800">
        <div className="max-w-7xl mx-auto px-4 sm:px-6 lg:px-8">
          <div className="flex items-center justify-between h-16">
            <div className="flex items-center">
              <div className="flex-shrink-0">
                <img className="h-8 w-32" alt="Workflow" />
              </div>
              <div className="hidden md:block">
                <div className="space-x-4">
...
```

So you will spend your time to align something in the proper way, change the icon, change the fading colors settings and so on. That's really not for me (as you can see from the below framework I wrote 14 years ago, really ugly from a stylish point of view), you won't find a complete tutorial here on this subject, I couldn't do it. I should have get my hands dirty on it, I did it for a few tests but not enough to really help people.

https://github.com/ricky-andre/Cisco-Config-Surfer-Parser

In case you are interested in going deeper, this a written on-line guide:
https://ibaslogic.com/react-tutorial-for-beginners/
... and this is a 12 hours long video from freecodecamp:
https://www.youtube.com/watch?v=bMknfKXIFA8

Did I convince you that it’s not that easy ? if not, here’s the content index of the above video. As usual, you don’t really need to know everything if you just wanna write dApps, but this is just to let you understand that behind every technology there’s a whole world. Only HR guys pretend not to know it, and even when they do, their managers wouldn’t. It’s the usual ‘insane-chain’. A blockchain, is always a better chain.

0:00 Introduction
5:27 What we’ll learn
7:03 Fun facts about react link: https://www.figma.com/file/xA1rJVQOor...
9:08 First react
https://reactjs.org/docs/cdn-links.html
17:13 First React Practice
19:04 Local Setup (the quick way)
21:03 Why React?
30:38 JSX
40:19 Goodbye, CDNs!
44:27 Thought Experiment
49:57 Project 1 Part 1 - MarkUp
57:44 Pop Quiz!
59:55 Components
1:33:07 Setup a local React environment w/ Create React App
1:33:53 Babel, Bundler, Build
1:34:47 Create React app: https://create-react-app.dev/
1:35:56 How to install Node.js
1:36:06 Use nvm or nvm-windows
1:36:33 How to install Node.js
1:41:30 Styles and images with CRA
https://create-react-app.dev/docs/add...
https://create-react-app.dev/docs/usi...
1:46:03 Quick Mental Outline of Project
1:50:00 Quick Figma Walkthrough
https://www.youtube.com/watch?v=ybc2g...
1:51:43 Project Setup
https://www.figma.com/file/xA1rJVQOor...
1:59:00 Navbar and Styling
2:06:18 Main Section
2:14:04 Color The Bullets
2:16:30 Add Background Logo
2:20:50 Section 1 Solo Project
2:22:23 Digital Business Card https://scrimba.com/links/figma-digit...
2:24:05 Share your work https://scrimba.com/links/solo-projec...
https://scrimba.com/links/discord-i-b...
2:24:45 Section 1 Recap
https://scrimba.com/links/discord-tod...

Build an AirBnb Experiences Clone
2:27:26 Section intro & Figma File
https://scrimba.com/links/figma-airbn...
2:31:40 Project Setup: NavBar & Hero
2:43:11 Project Card Component
2:50:32 Problem - Not Reusable
2:52:29 Props
3:18:42 Prop Quiz (Get it?)
3:23:10 Destructuring Props
3:27:05 Props practice
3:36:12 Passing in non-string Props
3:40:11 Project: Pass props to component
3:47:08 Review - Array.map()
3:55:37 React render array
4:00:10 Mapping Components
4:04:46 Map Quiz
4:08:26 Loading Images from .map()
4:10:02 Projects
4:32:34 Spread objects as props
4:36:30 Section 2 solo project
4:37:14 Travel journal: https://scrimba.com/links/figma-trave...
4:39:24 Share your work

4:39:52 Section 2 recap

Build a Meme Generator
4:41:37 Section into and figma file
https://scrimba.com/links/figma-meme-...
4:45:48 Meme Generator: Header & Form
4:57:13 Project Analysis
4:58:20 Event Listeners
5:04:31 Project: Get random meme
5:10:15 Our current conundrum
5:18:26 Props vs. State
5:32:13 useState
5:37:57 Changing State
5:41:03 useState - Counter Practice
5:45:51 useState - Changing state with a callback Function
5:51:12 hanging State Quiz!
5:53:44 Project: Ass images to the meme generator
5:56:43 Challenge: Ternary Practice & flipping State back and forth
6:06:37 Complex State
6:27:46 Project: Refactor State
6:31:59 Passing state as props
6:37:54 Setting state from child components
6:44:25 Passing data around
6:50:53 Boxes Challenge
7:28:46 Conditional Rendering
7:48:49 React forms intro
7:52:17 Watch for input changes in React
7:56:49 Form inputs practice
7:59:13 Forms state object
8:07:18 Controlled inputs
8:11:35 Forms in React
8:47:04 Project: add text to image
8:51:05 Making API Calls
8:55:08 Intro to useEffect
https://reactjs.org/docs/hooks-effect...
9:00:54 useEffect()
9:42:46 Project:get memes from API
9:33:00 State and Effect Practices
9:40:05 useEffect cleanup function
9:46:00 Using an sync function inside useEffect
9:49:14 Section3 recap

Build a Notes app and Tenzies Game
9:51:34 Section 4 Intro
https://scrimba.com/links/figma-react...
https://scrimba.com/links/figma-tenzi...
9:54:09 Warm-up:Add Dark/Light modes to ReactFacts Site
10:00:50 Notes App Intro
10:10:47 Notes App Development
10:44:17 Tenzies Project Intro
https://scrimba.com/links/figma-tenzi...
10:45:38 Tenzies Setup & Game Development
11:24:35 Hold dice part 3
11:28:39 End game
https://github.com/alampros/react-con...
11:40:31 Tenzies: New Game & Extra Credit ideas
11:44:15 Section 4 Solo Project
11:45:53 quiz https://scrimba.com/links/figma-quizz...
11:47:26 OTDB API https://opentdb.com/api_config.php
Check out the class components crash course: https://scrimba.com/playlist/pBpayAz
11:49:32 Congrats on completing Module 1!

10.1 Creating a React project and directory structure

To create a react project you can use the following command:

```
create-react-app <my project>
.
```

The following directories will be created:

```
├── README.md
├── node_modules
├── package.json
├── .gitignore
├── build
├── public
│   ├── favicon.ico
│   ├── index.html
│   └── manifest.json
└── src
    ├── App.css
    ├── App.js
    ├── App.test.js
    ├── index.css
    ├── index.js
    ├── logo.svg
    └── serviceWorker.js
```

build represents the path to our final production build. This folder would actually be created after we run the npm build.
We can see all the "**dependencies**" and "**devDependencies**" required by our React app in node_modules. These are as specified or seen in our **package.json** file.
Our **static files** are located in the **public** directory. Files in this directory will retain the same name when deployed to production. Thus, they can be cached at the client-side and improve the overall download times.
All of the **dynamic components** will be located in the **src**. To ensure that, at the client side, only the most recent version is downloaded and not the cached copy, Webpack will generally have the updated files a unique file name in the final build. Thus, we can use simple file names e.g. header.png, instead of using header-2019-01-01.png. Webpack would take care of renaming header.png to header.dynamic-hash.png. This unique hash would get updated only when our header.png would change. We can also see files like App.js which is kind of our main JS component and the corresponding styles go in App.css. In case, we want to add any unit tests, we can use the App.test.js for that. Also, **index.js** is the **entry point for our App** and it triggers the registerServiceWorker.js. As a side-note, we mostly add a 'components' directory here to add new components and their associated files, as that improves the organization of our structure.

The **overall configuration** for the React project is outlined in the **package.json**. Below is what that looks like:

```
{
        "name": "my-sample-app",
        "version": "0.0.1",
        "private": true,
        "dependencies": {
        "react": "^16.5.2",
            "react-dom": "^16.5.2",
        },
        "devDependencies": {
            "react-scripts": "1.0.7"
        },
        "scripts": {
            "start": "react-scripts start",
            "build": "react-scripts build",
            "test": "react-scripts test --env=jsdom",
            "eject": "react-scripts eject"
        }
}
```

We can see the following attributes:

- name - Represents the app name which was passed to create-react-app.
- version - Shows the current version.

- dependencies - List of all the required modules/versions for our app. By default, npm would install the most recent major version.
- devDependencies - Lists all the modules/versions for running the app in a development environment.
- scripts - List of all the aliases that can be used to access react-scripts commands in an efficient manner. For example, if we run npm build in the command line, it would run "react-scripts build" internally.

The dependencies which are shared by our application can go to the assets directory. These can include mixins, images, etc. Thus, they would represent a single location for files external to our main project itself. We also need to have a utilities folder. This would contain a list of helper functions used globally across the app. We can add common logic to this utilities folder and import that wherever we want to use it. While the naming can vary slightly, the standard naming conventions are seen include helpers, utils, utilities, etc.
With that, our structure would now looks something like below:

```
my-sample-app
├── build
├── node_modules
├── public
│   ├── favicon.ico
│   ├── index.html
│   └── manifest.json
├── src
│   ├── assets
│   │   └──images
│   │      └── logo.svg
│   ├── components
│   │   └── app
│   │       ├── App.css
│   │       ├── App.js
│   │       └── App.test.js
│   ├── utilities
│   ├── Index.css
│   ├── Index.js
│   └── service-worker.js
├── .gitignore
├── package.json
└── README.md
```

manifest.json This file is used to describe our app e.g. On mobile phones, if a shortcut is added to the home screen. Below is how that would look like:

```
{
  "short_name": "My Sample React App",
  "name": "My Create React App Sample",
  "icons": [
    {
      "src": "favicon.ico",
      "sizes": "64x64 32x32 24x24 16x16",
      "type": "image/x-icon"
    }
  ],
  "start_url": ".",
  "display": "standalone",
  "theme_color": "#efefef",
  "background_color": "#000000"
}
```

When our web app is added to user's home screen, it is this metadata which determines the icon, theme colors, names, etc.

10.2 React with Vite

There is another and more simple way to create a react project:

https://www.youtube.com/watch?v=Wn_Kb3MR_cU&t=446s

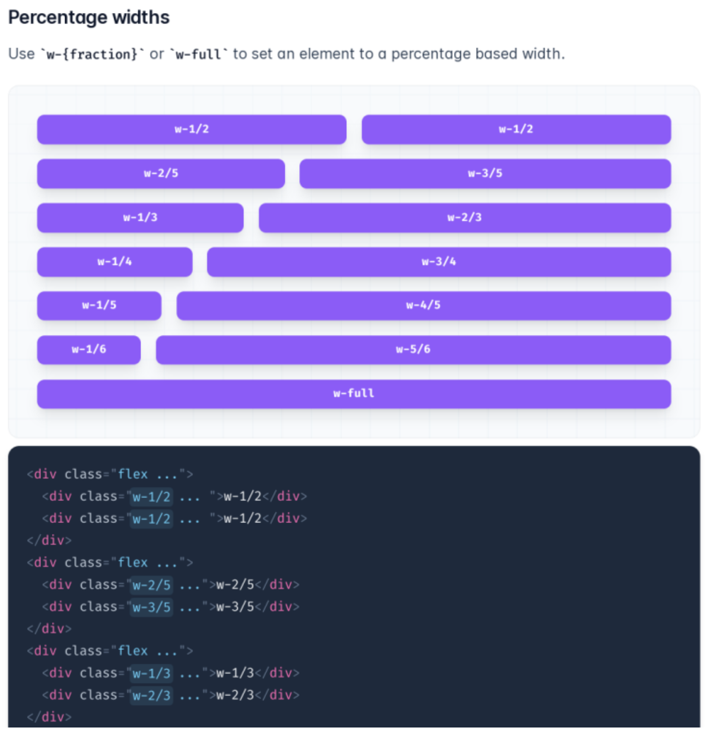
```
npm init vite@latest
```

You will be prompted for the following information:

```
Project name:     <project name> (can be ./ if you're already in the right folder)
Package name:     <package name>
Framework:        react

! in general, this command must be run inside the root directory of the project
! to locally install all project's dependencies, which can be easily 150-300MB of data.
! You start probably understanding the advantages of using dockers and containers for
! software developers.
npm install
npm run dev
```

The last command will start a browser listening on port 3000. **Tailwindcss** is another tool to build UI without having to write CSS (Cascaded Style Sheet).

```
npm install -D tailwindcss postcss autoprefixer
npx tailwindcss init -p

! after having copied some stuff locally
npm run start
npm run dev
```



Percentage widths

Use `w-{fraction}` or `w-full` to set an element to a percentage based width.

```
<div class="flex ...">
  <div class="w-1/2 ... ">w-1/2</div>
  <div class="w-1/2 ... ">w-1/2</div>
</div>
<div class="flex ...">
  <div class="w-2/5 ...">w-2/5</div>
  <div class="w-3/5 ...">w-3/5</div>
</div>
<div class="flex ...">
  <div class="w-1/3 ...">w-1/3</div>
  <div class="w-2/3 ...">w-2/3</div>
</div>
```

Documentation can be found here:
https://tailwindcss.com/docs/display#flex
... the previous picture was about the 'w-full' feature.

10.3 Component Directory

The component directory structure is the most important thing in any React app. While components can reside in src/components/my-component-name, it is recommended to have an **index.js** inside that directory.

Thus, whenever someone imports the component using src/components/my-component-name, instead of importing the directory, this would actually import the index.js file.

Also component involves many files, including stateless and stateful containers, SASS files, utilities shared within that component, and even child components.

Thus, our component directory structure would look something like below:

```
my-sample-app
└── src
      └── components
       └── my-component-name
                       ├── my-component-name.css
           ├── my-component-name.scss
           ├── my-component-name-container.js
           ├── my-component-name-redux.js
           ├── my-component-name-styles.js
            ├── my-component-name-view.js
            └── index.js
```

My-component-name.css represents the CSS file imported by our stateless view Component. My-component-name.scss is the SASS file imported by our stateless view Component. My-component-name-container.js would contain the business logic as well as state management. My-component-name-redux.js would include mapStateToProps, mapDispatchToProps and connect functionality provided by Redux. My-component-name-styles.js would represent our JSS (e.g. storing Material UI styles). My-component-name-view.js would mostly be a pure functional Component index.js is the entry point for importing our Component.

10.4 Unit Tests

For unit tests, we would follow the same principle of grouping all our related files. Thus, we can add them within the components directory we have as shown below;

```
my-sample-app
└── src
    └── components
        └── my-component-name
            ├── my-component-name-container.js
            ├── my-component-name-container.test.js
            ├── my-component-name-redux.js
            ├── my-component-name-redux.test.js
```

Beware that testing is NOT an option, in general but especially on blockchains. Almost always there are real money that can be stolen if things are not properly done, thus automatic and EXTENSIVE testing is fundamental and not an option.

10.5 Index Page

Let's also have a look inside the index.js as well as the index.html page which gets generated.

Below is how our index.js file looks;

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import registerServiceWorker from './registerServiceWorker';

ReactDOM.render(<App />, document.getElementById('root'));
registerServiceWorker();
```

Below is the html page;

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta
      name="viewport"
      content="width=device-width, initial-scale=1, shrink-to-fit=no"
    />
     <meta name="theme-color" content="#000000" />
     <!--
       manifest.json provides metadata used when your web app is installed on a
       user's mobile device or desktop. See https://developers.google.com/web/fundamentals/web-app-manifest/
     -->
     <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
     <!--
       Notice the use of %PUBLIC_URL% in the tags above.
       It will be replaced with the URL of the `public` folder during the build.
       Only files inside the `public` folder can be referenced from the HTML.

       Unlike "/favicon.ico" or "favicon.ico", "%PUBLIC_URL%/favicon.ico" will
       work correctly both with client-side routing and a non-root public URL.
       Learn how to configure a non-root public URL by running `npm run build`.
     -->
     <title>React App</title>
   </head>
   <body>
     <noscript>You need to enable JavaScript to run this app.</noscript>
     <div id="root"></div>
     <!--
       This HTML file is a template.
       If you open it directly in the browser, you will see an empty page.

       You can add webfonts, meta tags, or analytics to this file.
       The build step will place the bundled scripts into the <body> tag.

       To begin the development, run `npm start` or `yarn start`.
       To create a production bundle, use `npm run build` or `yarn build`.
     -->
   </body>
 </html>
```

As we can see, that it is a very basic HTML page with a few meta tags and some link elements. Also, we can see that there is an empty div element which is added with id "root". We can always update that to something else, like "content", as well as add any additional CSS or external JS libraries e.g. say we want to add Bootstrap library to our project. To do that, we can directly add a CDN reference to our index.html, as shown below:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta
      name="viewport"
      content="width=device-width, initial-scale=1, shrink-to-fit=no"
    />
    <meta name="theme-color" content="#000000" />
    <!--
      manifest.json provides metadata used when your web app is installed on a
      user's mobile device or desktop. See https://developers.google.com/web/fundamentals/web-app-manifest/
    -->
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <!--
      Notice the use of %PUBLIC_URL% in the tags above.
      It will be replaced with the URL of the `public` folder during the build.
      Only files inside the `public` folder can be referenced from the HTML.

      Unlike "/favicon.ico" or "favicon.ico", "%PUBLIC_URL%/favicon.ico" will
      work correctly both with client-side routing and a non-root public URL.
      Learn how to configure a non-root public URL by running `npm run build`.
    -->
    <title>React App</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="content"></div>
    <!--
      This HTML file is a template.
      If you open it directly in the browser, you will see an empty page.

      You can add webfonts, meta tags, or analytics to this file.
      The build step will place the bundled scripts into the <body> tag.

      To begin the development, run `npm start` or `yarn start`.
      To create a production bundle, use `npm run build` or `yarn build`.
    -->
     <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></script>

<script src="https://code.jquery.com/jquery-3.2.1.slim.min.js" crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.11.0/umd/popper.min.js" crossorigin="anonymous"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta/js/bootstrap.min.js" crossorigin="anonymous"></script>
  </body>
</html>
```

Since we changed the default container element Id to "content", we also have to update the same in our index.js as shown below:

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import registerServiceWorker from './registerServiceWorker';

ReactDOM.render(<App />, document.getElementById('content'));
registerServiceWorker();
```

The other way of adding the bootstrap library (say we want to use it only within a specific component) is to use npm to install the library and then add the import as shown below:

```
npm install --save bootstrap
import '../node_modules/bootstrap/dist/css/bootstrap.min.css';
import 'bootstrap/dist/css/bootstrap.min.css';
```

10.6 Ejecting

Let's say that, after you generated the project using create-react-app, you want to do some additional customization. Ejecting would allow you to do that.
Following command can be run to eject from create-react-app:

```
npm eject
```

Ejecting would mean that all the configuration gets exposed to us and we would be responsible for maintaining all the configuration from that point onward.
Thus, it essentially allows us more control over the project. It is important to remember that this is a on-way command i.e. we cannot go back once we eject.

10.7 Building, debugging, running the project

The following command will download and install all project dependencies and libraries that have been listed in the **package.json** file.

```
yarn install
```

The following command will locally start a web server listening on port 3000, you need to be in the main directory to launch this command, usually 'src'.

```
yarn start
```

To create a production build use the following command:

```
yarn bild
```

10.8 Connecting Metamask Wallet

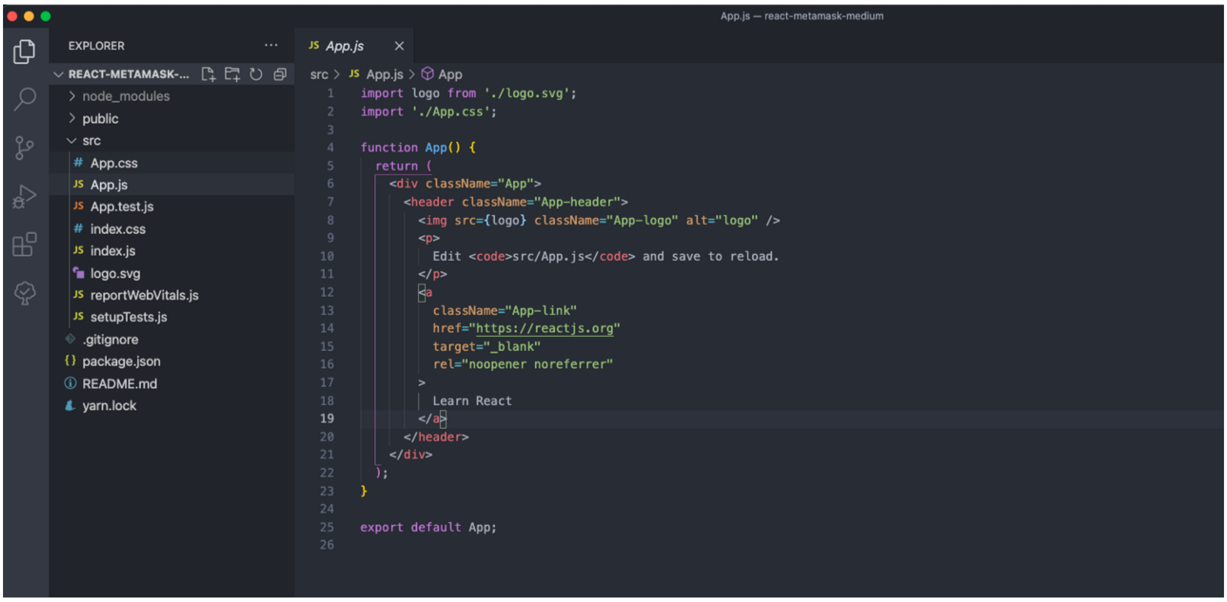
There are a few tutorials on the web, some of them providing github content. You clone them, install those 150MB of libraries, launch them, and they don't work. Probably it's to show they did 'something', nobody's gonna really check if it works or not.

https://medium.com/coinmonks/connecting-to-metamask-react-js-custom-hook-state-management-2f1f3203f509

Let's first start by creating a new app with React. I'm using `npx create-react-app` . I am also using VS Code for this tutorial as well just to note.

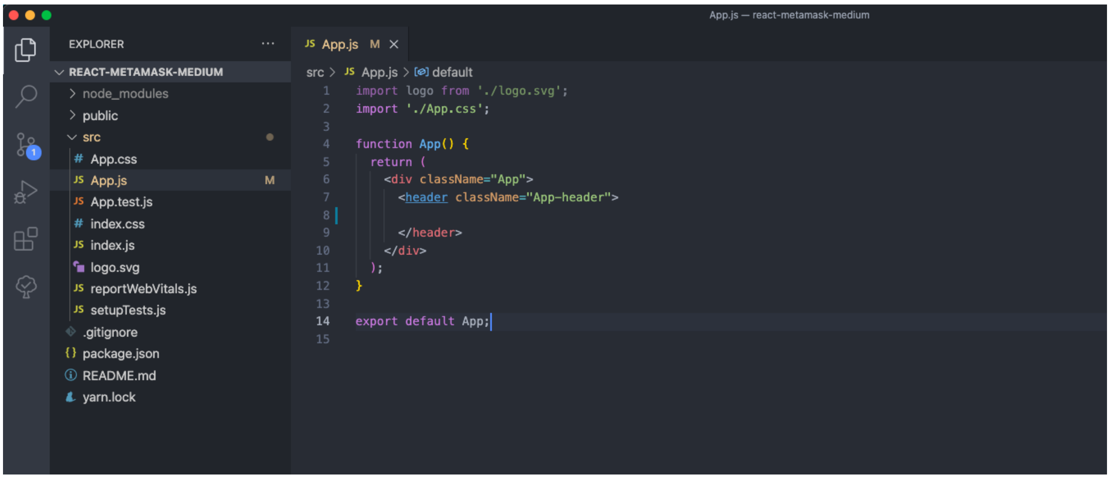
```
npx create-react-app react-metamask-medium && cd react-metamask && code .
```

Ok, good to go! You should have a simple react application, and if you open up **App.js** it should look like this:



```
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

Let's do a little cleaning and remove everything we don't need. I am going to remove everything in between `<header></header>` and place my new content in there. I am going to be using the existing CSS to center and drop the new content:

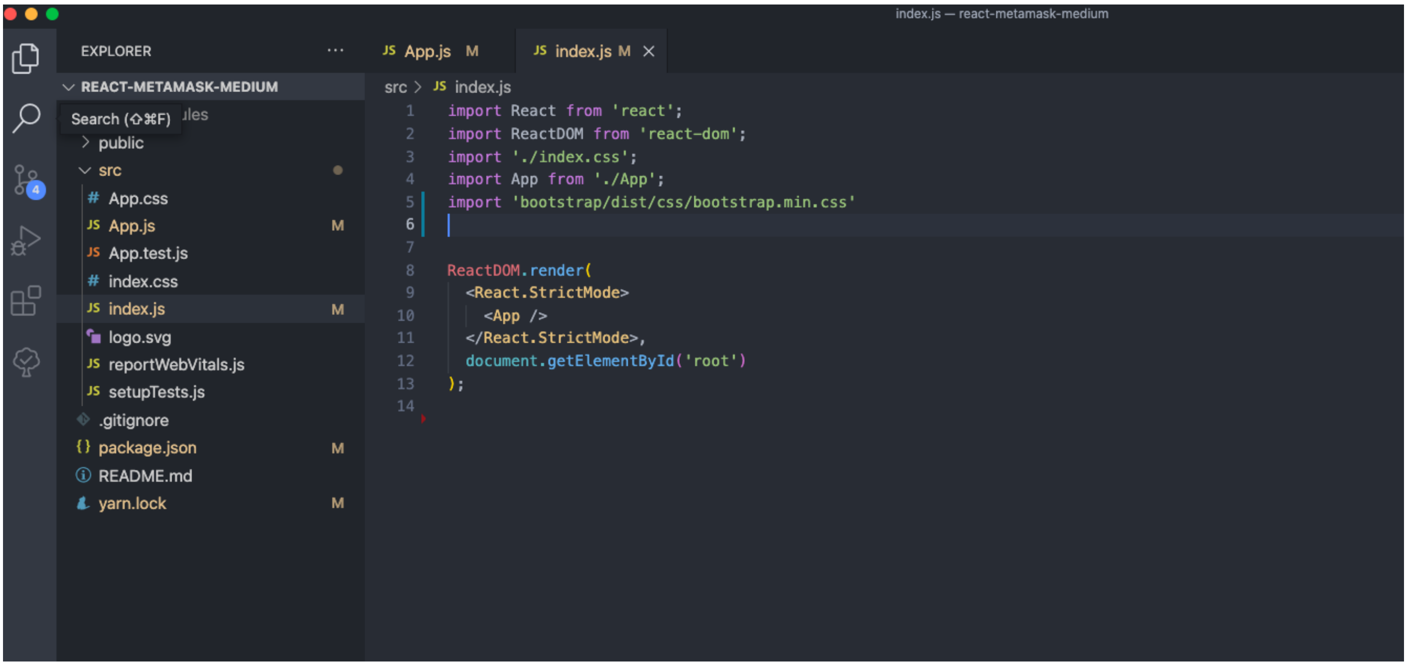


```
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">

      </header>
    </div>
  );
}

export default App;
```

Now let’s add a few package to drop in some buttons for the content. I am also going to add a MetaMask logo and hand wave svg, here is the link to download if you’re interested: metamask.svg. hand.svg

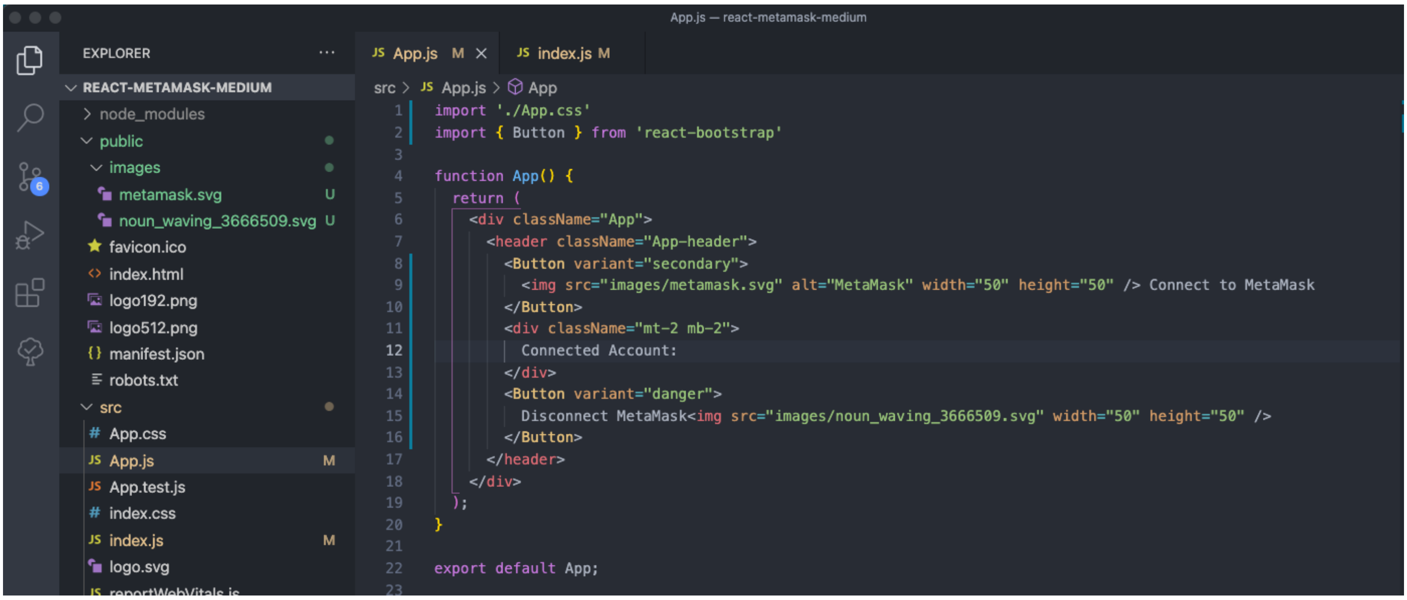
```
yarn add react-bootstrap bootstrap@5.1.3
```

Let’s add `import ‘bootstrap/dist/css/bootstrap.min.css’` to our **index.js** file and also remove some extra code. Your index.js should look like this:



```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import 'bootstrap/dist/css/bootstrap.min.css'


ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);

```

Now let’s add the buttons to the app so we can connect to MetaMask.



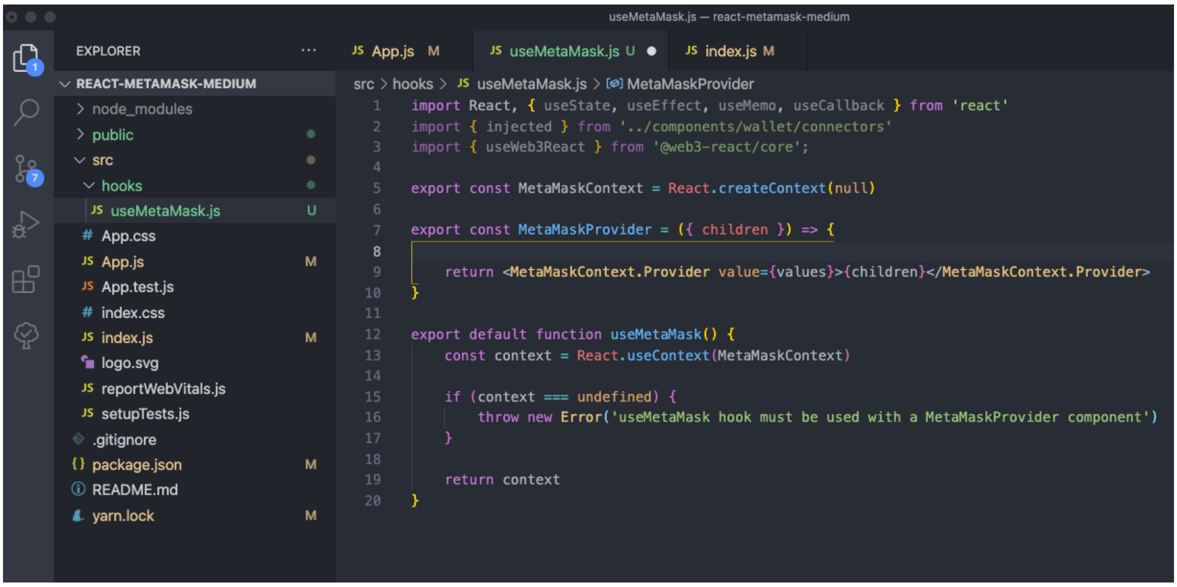
```
import './App.css'
import { Button } from 'react-bootstrap'

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <Button variant="secondary">
          <img src="images/metamask.svg" alt="MetaMask" width="50" height="50" /> Connect to MetaMask
        </Button>
        <div className="mt-2 mb-2">
          Connected Account:
        </div>
        <Button variant="danger">
          Disconnect MetaMask<img src="images/noun_waving_3666509.svg" width="50" height="50" />
        </Button>
      </header>
    </div>
  );
}

export default App;
```

Now we need to create a Custom Hook that we can use to call connect and disconnect on these buttons. This custom hook will also keep track of the state of our app and dish out any data we need such as **account info** or **balance**.

We should create a folder called **hooks** and drop in **useMetaMask.js** for the custom hook, so our folder structure looks like this: **src/hooks/useMetaMask.js**

To keep track of the State of this hook throughout the entire app's lifecycle, we'll have to utilize React's **createContext**

This is a bit much to break down, but for a simple boiler plate that we will fill, please see the following code:



```
import React, { useState, useEffect, useMemo, useCallback } from 'react'
import { injected } from '../components/wallet/connectors'
import { useWeb3React } from '@web3-react/core';

export const MetaMaskContext = React.createContext(null)

export const MetaMaskProvider = ({ children }) => {

    return <MetaMaskContext.Provider value={values}>{children}</MetaMaskContext.Provider>
}

export default function useMetaMask() {
    const context = React.useContext(MetaMaskContext)

    if (context === undefined) {
        throw new Error('useMetaMask hook must be used with a MetaMaskProvider component')
    }

    return context
}
```

For this component we are going to need the following from React, **useState, useEffect, useMemo, and useCallback**

We are also going to need to use @web3-react/core's **useWeb3React** hook as well as create a new component called **injected.** This injected component will utilize web3-react/injected-connector to connect supported chain's to the app and MetaMask. Let's run the following:

```
yarn add @web3-react/injected-connector @web3-react/core
```

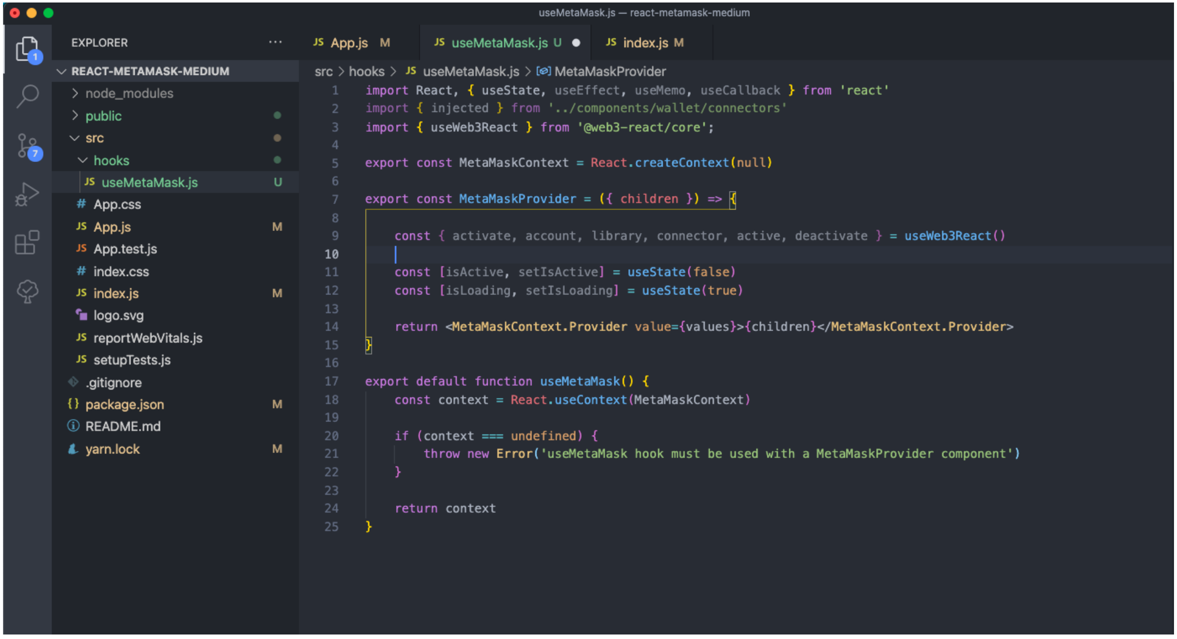
Next let's begin to fill out our new useMetaMask hook to work with out app. To do this we will drop all our code inside the **MetaMaskProvider** component.

First up we will want to add the useWeb3React hook to gather some important resources for connecting with MetaMask.

```
const { activate, account, library, connector, active, deactivate } = useWeb3React()
```

I also start to create some specific states I want to keep track of like **isActive** and **isLoading.**
isLoading will be useful to tell when the useMetaMask hook is loading to read its connection with MetaMask. **isActive** will be useful to tell if MetaMask is currently connected to the app with the proper chain i am allowing inside the app.

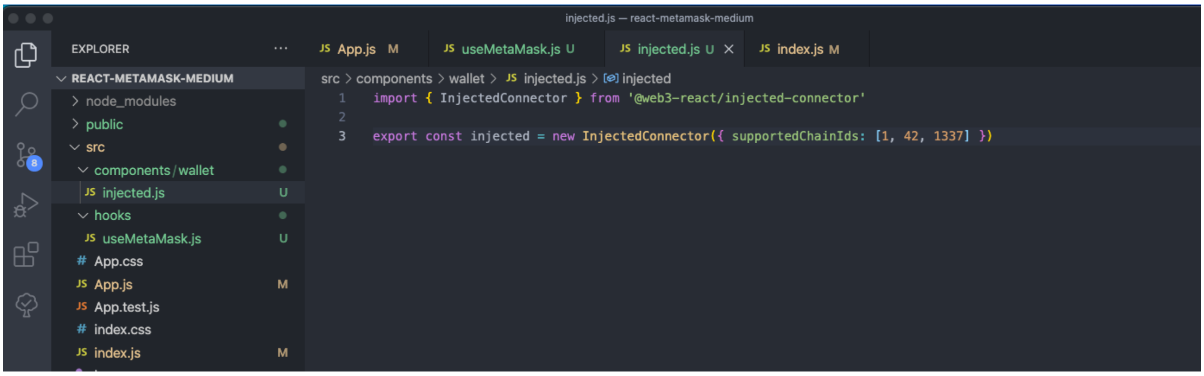
```
const [isActive, setIsActive] = useState(false)
const [isLoading, setIsLoading] = useState(true)
```



```
import React, { useState, useEffect, useMemo, useCallback } from 'react'
import { injected } from '../components/wallet/connectors'
import { useWeb3React } from '@web3-react/core';

export const MetaMaskContext = React.createContext(null)

export const MetaMaskProvider = ({ children }) => {

    const { activate, account, library, connector, active, deactivate } = useWeb3React()

    const [isActive, setIsActive] = useState(false)
    const [isLoading, setIsLoading] = useState(true)

    return <MetaMaskContext.Provider value={values}>{children}</MetaMaskContext.Provider>
}

export default function useMetaMask() {
    const context = React.useContext(MetaMaskContext)

    if (context === undefined) {
        throw new Error('useMetaMask hook must be used with a MetaMaskProvider component')
    }

    return context
}
```

To see the full code on this custom hook please click here
Next I should create my injected component which will be used to connect MetaMask to specific the specific Chains I am wanting to use with my app.
I will create a folder called **src/components/wallet/injected.js** to drop the injected component file into. The code is simple:

```
import { InjectedConnector } from '@web3-react/injected-connector'
export const injected = new InjectedConnector({ supportedChainIds: [1, 42, 1337] })
```



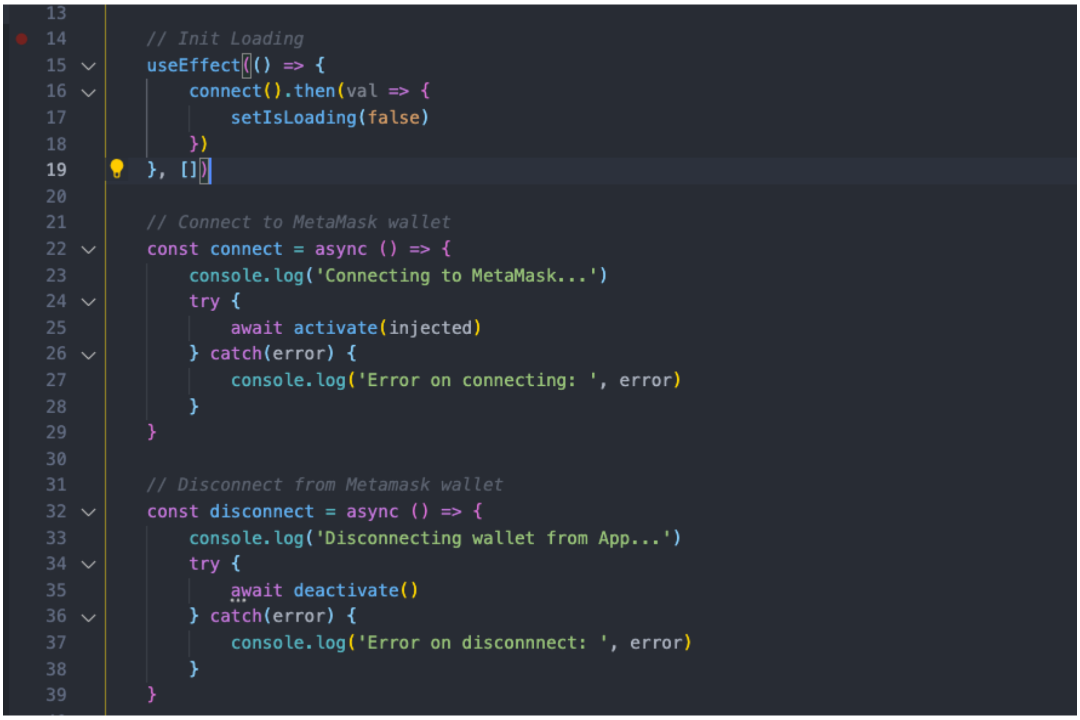
```
import { InjectedConnector } from '@web3-react/injected-connector'

export const injected = new InjectedConnector({ supportedChainIds: [1, 42, 1337] })
```

The **supportedChainIds** will help to make sure MetaMask is connected to the proper chains my app is using, otherwise it won't show active in our custom hook.
For more support Chain IDs please see https://chainlist.org/

1 = Ethereum Mainnet
42 = Kovan Testnet — Which I will use to connect in a later article using Web3 and Infura which will come in handing is making simple simple transactions to test out my app
1337 = Local Host chain. For this I used Ganache to create a local chain and connect to my MetaMask wallet.

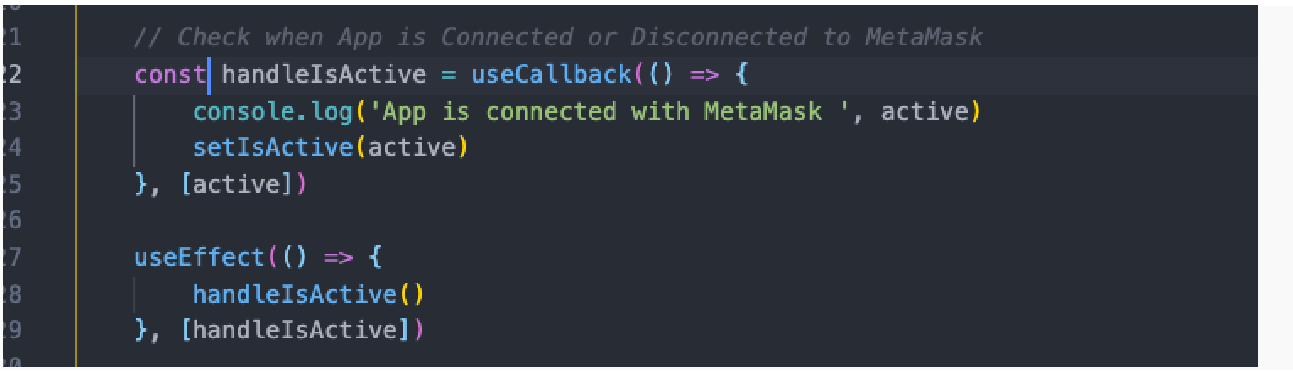
Now back to our **useMetaMask** hook.

We want to use **useEffect** with no dependency which will run once when the hook is initialized. Inside of this **useEffect** hook will will drop on a **connect()** function to initialize the connection to the App and MetaMask when the app is first ran:



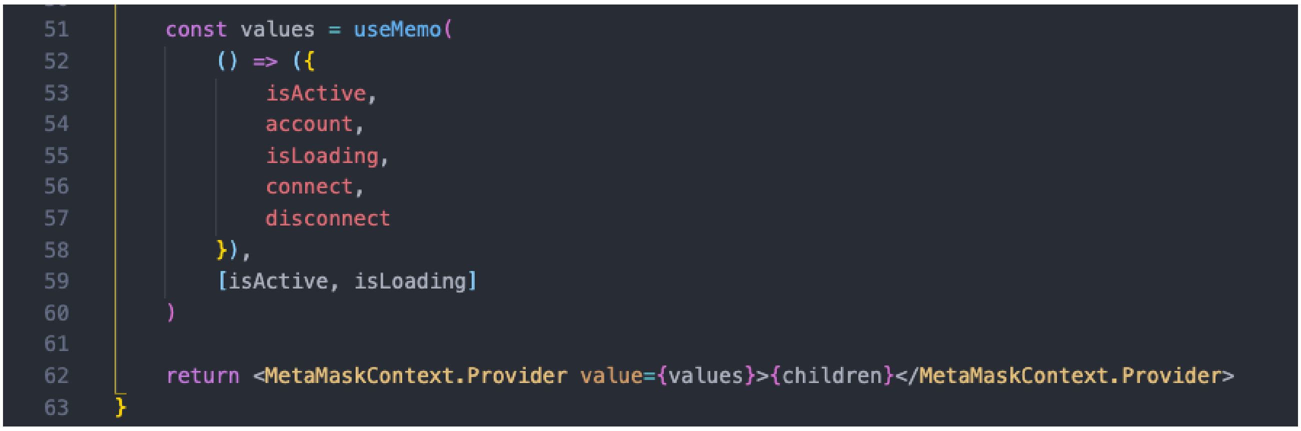
```
// Init Loading
useEffect(() => {
    connect().then(val => {
        setIsLoading(false)
    })
}, [])

// Connect to MetaMask wallet
const connect = async () => {
    console.log('Connecting to MetaMask...')
    try {
        await activate(injected)
    } catch(error) {
        console.log('Error on connecting: ', error)
    }
}

// Disconnect from Metamask wallet
const disconnect = async () => {
    console.log('Disconnecting wallet from App...')
    try {
        await deactivate()
    } catch(error) {
        console.log('Error on disconnnect: ', error)
    }
}
```

These functions when called will help connect and disconnect the wallet to the app. I am using them as a useCallback function because I don't want them to re-render more than needed in the app, only when they are called.
Next let's create a **handleIsActive** function to check if MetaMask is currently connected to our app. I am going to use a **useCallback** for this and drop in **active** property from our **useWeb3React** hook as a dependency to update our hook when MetaMask is connected and disconnected from our app. This will usually be because the user switch accounts on MetaMask for a chain that was not connected to our app:



```
// Check when App is Connected or Disconnected to MetaMask
const handleIsActive = useCallback(() => {
    console.log('App is connected with MetaMask ', active)
    setIsActive(active)
}, [active])

useEffect(() => {
    handleIsActive()
}, [handleIsActive])
```

I also created a useEffect hook that will depend on **handleIsActive** and run this function only when **active** has changed inside the **handleIsActive** callback. This will update our app anytime it becomes **active** true or false. Disconnecting MetaMask from our app will also cause **active** to be false. Last to make sure that the useMetaMask custom hook updates the rest of our app accordingly if any of its dependencies change, we will use **useMemo** to change the props of our hook accordingly. We do this so the rest of the app can tell if **isActive** or **isLoading** is updated. Also this will give access to our **connect** and **disconnect** functions, plus our **account.** They will only reevaluate when our needed dependencies change such as **isActive** and **isLoading**:

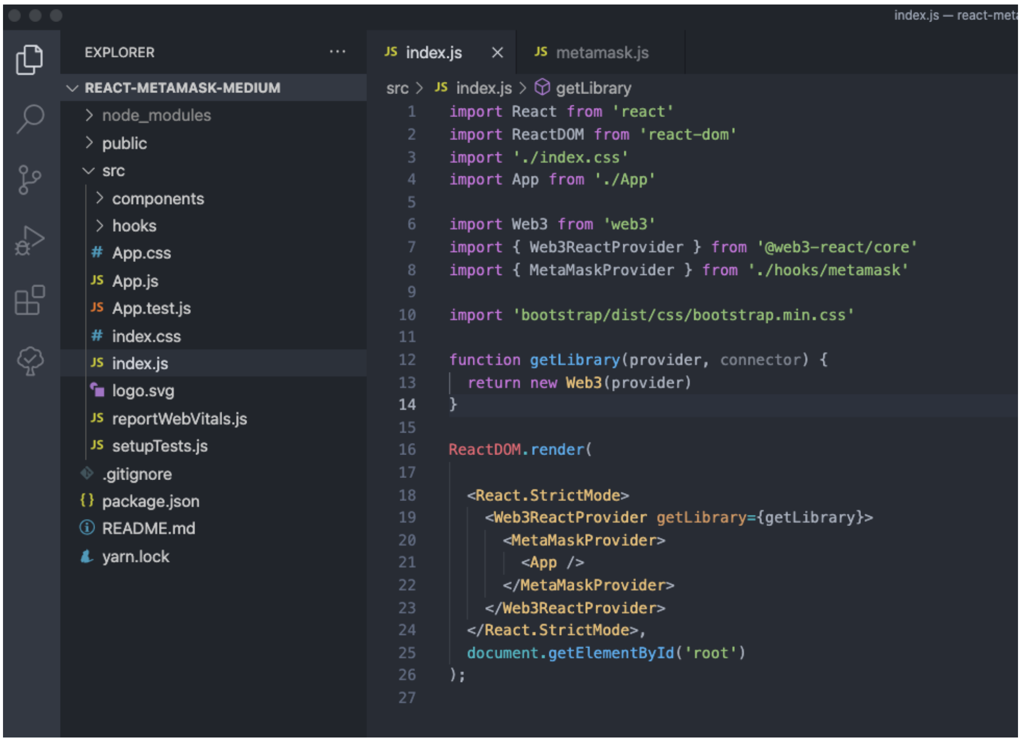


```
    const values = useMemo(
        () => ({
            isActive,
            account,
            isLoading,
            connect,
            disconnect
        }),
        [isActive, isLoading]
    )

    return <MetaMaskContext.Provider value={values}>{children}</MetaMaskContext.Provider>
}
```

Cool now good to go. To checkout out the full source code of this custom hook click here
Last step is to use the **Web3ReactProvider** around our existing app and also use our **MetaMaskProvider** as well to wrap the app so we can have the state of our custom hook existing at all times.
We need to lastly add our last dependency:

```
yarn add web3
```

Once we wrap the app in **index.js** and use everything it should look like this:



```
import React from 'react'
import ReactDOM from 'react-dom'
import './index.css'
import App from './App'

import Web3 from 'web3'
import { Web3ReactProvider } from '@web3-react/core'
import { MetaMaskProvider } from './hooks/metamask'

import 'bootstrap/dist/css/bootstrap.min.css'

function getLibrary(provider, connector) {
  return new Web3(provider)
}

ReactDOM.render(

  <React.StrictMode>
    <Web3ReactProvider getLibrary={getLibrary}>
      <MetaMaskProvider>
        <App />
      </MetaMaskProvider>
    </Web3ReactProvider>
  </React.StrictMode>,
  document.getElementById('root')
);
```

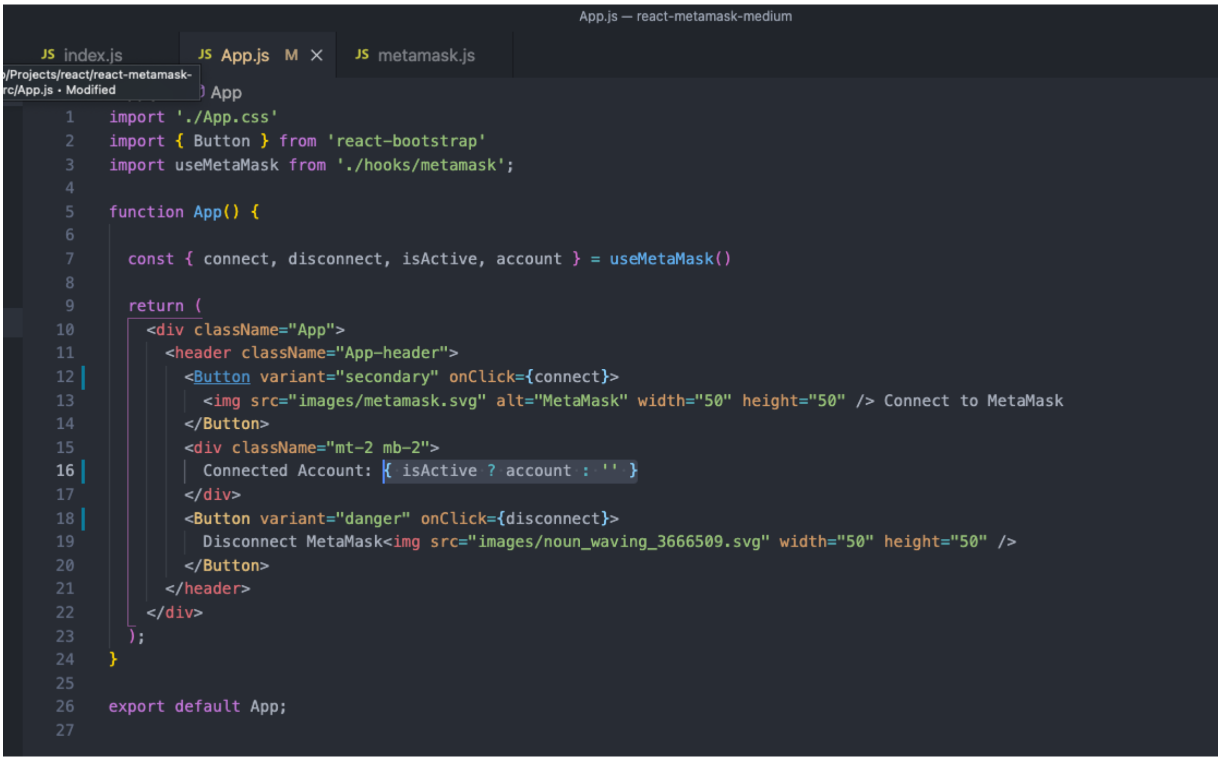
The reason for wrapping out app is the following **Web3ReactProvider** is what our **useWeb3React** hook depends on and it also paves the way for using **Web3** and transacting with our app.
Using the **MetaMaskProvider** around our <App /> gives us the ability to keep the state of our custom hook throughout the entire app. Next we can pop back to our **App.js** and use our Custom Hook we created.
Drop in the following to our App:

```
const { connect, disconnect, isActive, account } = useMetaMask()
```

As well we should put the following next to our Connected Account:

```
{ isActive ? account : '' }
```

This is going to tell our app if the account is connected, then show the account info. If not show a blank string.
We should also connect our buttons to connect and disconnect functions from our useMetaMask hook.
The end result will look like the following:


App.js — react-metamask-medium
JS index.js
JS App.js M ×
JS metamask.js
/Projects/react/react-metamask-
rc/App.js • Modified
App

```
import './App.css'
import { Button } from 'react-bootstrap'
import useMetaMask from './hooks/metamask';

function App() {

  const { connect, disconnect, isActive, account } = useMetaMask()

  return (
    <div className="App">
      <header className="App-header">
        <Button variant="secondary" onClick={connect}>
          <img src="images/metamask.svg" alt="MetaMask" width="50" height="50" /> Connect to MetaMask
        </Button>
        <div className="mt-2 mb-2">
          Connected Account: { isActive ? account : '' }
        </div>
        <Button variant="danger" onClick={disconnect}>
          Disconnect MetaMask<img src="images/noun_waving_3666509.svg" width="50" height="50" />
        </Button>
      </header>
    </div>
  );
}

export default App;

```

10.9 Web3.js

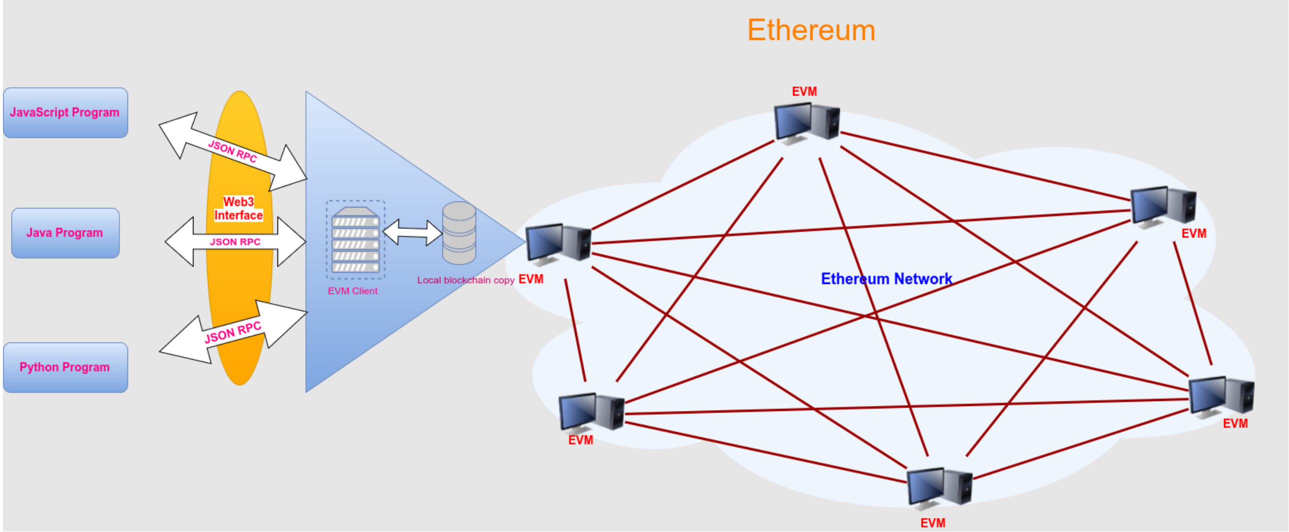
As usual the official documentation is a good point to start, and always a reference when you go on developing Dapps.

https://web3js.readthedocs.io/en/v1.7.1/getting-started.html#adding-web3-js

To create an object that can interact with a blockchain, we use the following:

```
var Eth = require('web3-eth');
// "Eth.providers.givenProvider" will be set if in an Ethereum supported browser.
var eth = new Eth(Eth.givenProvider || 'ws://some.local-or-remote.node:8546');
// ‘ws://localhost:8545’ for example using a local Ganache blockchain

// or using the web3 umbrella package
var Web3 = require('web3');
var web3 = new Web3(Web3.givenProvider || 'ws://some.local-or-remote.node:8546');
```

The following image has been taken from here:
https://iotbl.blogspot.com/2017/03/ethereum-and-blockchain-2.html
... while this intro to Web3.js has been taken from here:
https://www.dappuniversity.com/articles/web3-js-intro



We can get a JavaScript representation of an Ethereum smart contract with the `web3.eth.Contract()` function. This function expects two arguments: one for the smart contract ABI and one for the smart contract address. A smart contract ABI stands for "Abstract Binary Interface", and is a JSON array that describes how a specific smart contract works, in all details.
While we're here, I'll go ahead and store the address to the OMG token from the Ethereum main net:

```
const address = "0xd26114cd6EE289AccF82350c8d8487fedB8A0C07"
```

Now that we have both of these values assigned, we can create a complete JavaScript representation of the OMG token smart contract like this:

```
const contract = new web3.eth.Contract(abi, address)
```

Contract.methods returns a list of all the available (and PUBLIC or external) functions of a smart contract.

```
contract.methods.totalSupply().call((err, result) => { console.log(result) })
```

10.9.1 Building a transaction

You can create ‘raw transactions’, that need to be signed from the transaction’s sender private key. This is something that can be automated if you use tools like Brownie or Hardhat, but you should at least once get your hands dirty to understand what happens under the hood.
In addition to learning Web3.js, the purpose of this lesson is to help you understand the fundamentals about how transactions work on The Ethereum Blockchain. Whenever you create a transaction, you're writing data to the blockchain and updating its state. There are several ways to do this, like sending Ether from one account to another, calling a smart contract function that writes data, and deploying a smart contract to the blockchain. We can get a greater understanding of these concepts by performing these actions with the Web3.js library and observing how each step works.
In order to broadcast transactions to the network, we'll need to sign them first. I'm going to use an additional JavaScript library to do this called ethereumjs-tx. You can install this dependency from the command line like this:

```
$ npm install ethereumjs-tx
```

The reason we're going to use this library is that we want to **sign all of the transactions locally**. If we were running our own Ethereum node locally, we could unlock an account that was stored locally and sign all of our transactions locally. If that were the case, we would not necessarily need to use this library. However, we're using a remote node hosted by Infura in this tutorial. While Infura is a trustworthy service, we still want to sign the transactions locally rather than giving the remote node manage our private keys.

That's exactly what we'll do in this lesson. I'll show you how to create the raw transaction, sign it, then send the transaction and broadcast it to the network! In order to do this, I'm going to create a simple app.js file to run the code in this lesson, rather than doing everything in the console.

Inside the app.js file, we'll first require the newly installed library like this:

```
var Tx = require('ethereumjs-tx')
```

Next, we'll set up a Web3 connection like we did in the previous lessons:

```
const Web3 = require('web3')
const web3 = new Web3('https://ropsten.infura.io/YOUR_INFURA_API_KEY')
```

In this lesson, we're going to create a transaction that sends fake Ether from one account to another. In order to do this, we'll need two accounts and their private keys. You can actually create new accounts with Web3.js like this:

```
web3.eth.accounts.create()
{
    address: "0xb8CE9ab6943e0eCED004cDe8e3bBed6568B2Fa01",
    privateKey: "0x348ce564d427a3311b6536bbcff9390d69395b06ed6c486954e971d960fe8709",
    signTransaction: function(tx){...},
    sign: function(data){...},
    encrypt: function(password){...}
}
```

Once you have created both of these accounts, make sure you load them up with fake Ether from a faucet. Now, we'll assign them to variables in our script like this:

```
const account1 = '0xb8CE9ab6943e0eCED004cDe8e3bBed6568B2Fa01'
const account2 = '0xb8CE9ab6943e0eCED004cDe8e3bBed6568B2Fa02'
```

Be sure to use the accounts you generated, as these accounts won't work for this lesson. Now, let's save the private keys to the environment like this:

```
export PRIVATE_KEY_1='your private key 1 here'
export PRIVATE_KEY_1='your private key 2 here'
```

We want to save these private keys to our environment so that we don't hard code them into our file. It's bad practice to expose private keys like that. What if we accidentally committed them to source in a real project? Someone could steal our Ether! Now we want to read these private keys from our environment and store them to variables. We can do this with the process global object in NodeJS like this:

```
const privateKey1 = process.env.PRIVATE_KEY_1
const privateKey2 = process.env.PRIVATE_KEY_2
```

In order to sign transactions with the private keys, we must convert them to a string of binary data with a Buffer, a globally available module in NodeJS. We can do that like this:

```
const privateKey1 = Buffer.from(process.env.PRIVATE_KEY_1)
const privateKey1 = Buffer.from(process.env.PRIVATE_KEY_2)
```

Alright, now we've got all of our variables set up! I know some of this might be a little confusing at this point. Stick with me; it will all make sense shortly. :) You can also reference the video above if you get stuck. From this point, we want to do a few things:

- Build a transaction object
- Sign the transaction
- Broadcast the transaction to the network

We can build the transaction object like this:

```
const txObject = {
    nonce:    web3.utils.toHex(txCount),
    to:       account2,
    value:    web3.utils.toHex(web3.utils.toWei('0.1', 'ether')),
    gasLimit: web3.utils.toHex(21000),
    gasPrice: web3.utils.toHex(web3.utils.toWei('10', 'gwei'))
  }
```

Let me explain this code. We're building an object that has all the values needed to generate a transaction, like nonce, to, value, gasLimit, and gasPrice. Let's break down each of these values:

- **nonce** - this is the previous transaction count for the given account. We'll assign the value of this variable momentarily. We also must convert this value to hexidecimal. We can do this with the Web3.js utilitly web3.utils.toHex()
- **to** - the account we're sending Ether to
- **value** - the amount of Ether we want to send. This value must be expressed in Wei and converted to **hexidecimal**. We can convert the value to we with the Web3.js utility web3.utils.toWei().
- **gasLimit** - this is the maximum amount of gas consumed by the transaction. A basic transaction like this always costs 21000 units of gas, so we'll use that for the value here.
- **gasPrice** - this is the amount we want to pay for each unit of gas. I'll use 10 Gwei here.

Note, that there is no from field in this transaction object. That will be inferred whenever we sign this transaction with account1's private key.

Now let's get assign the value for the nonce variable. We can get the transaction nonce with web3.eth.getTransactionCount() function. We'll wrap all of our code inside a callback function like this:

```
web3.eth.getTransactionCount(account1, (err, txCount) => {
  const txObject = {
    nonce:    web3.utils.toHex(txCount),
    to:       account2,
    value:    web3.utils.toHex(web3.utils.toWei('0.1', 'ether')),
    gasLimit: web3.utils.toHex(21000),
    gasPrice: web3.utils.toHex(web3.utils.toWei('10', 'gwei'))
  }
})
```

And there is the completed transaction object! We've completed step 1. :) Now we must move on to step 2 where we sign the transaction. We can do that like this:

```
const tx = new Tx(txObject)
tx.sign(privateKey1)
const serializedTx = tx.serialize()
const raw = '0x' + serializedTx.toString('hex')
```

Here we're using the etheremjs-tx library to create a new Tx object. We also use this library to sign the transaction with privateKey1. Next, we serialize the transaction and convert it to a hexidecimal string so that it can be passed to Web3.Finally, we send this signed serialized transaction to the test network with the web3.eth.sendSignedTransaction() function like this:

```
web3.eth.sendSignedTransaction(raw, (err, txHash) => {
  console.log('txHash:', txHash)
})
```

And there you go! That's the final step of this lesson that sends the transaction and broadcasts it to the network. At this point, your completed `app.js` file should look like this:

```
var Tx     = require('ethereumjs-tx')
const Web3 = require('web3')
const web3 = new Web3('https://ropsten.infura.io/YOUR_INFURA_API_KEY')

const account1 = '' // Your account address 1
const account2 = '' // Your account address 2

const privateKey1 = Buffer.from('YOUR_PRIVATE_KEY_1', 'hex')
const privateKey2 = Buffer.from('YOUR_PRIVATE_KEY_2', 'hex')

web3.eth.getTransactionCount(account1, (err, txCount) => {
  // Build the transaction
  const txObject = {
    nonce:    web3.utils.toHex(txCount),
    to:       account2,
    value:    web3.utils.toHex(web3.utils.toWei('0.1', 'ether')),
    gasLimit: web3.utils.toHex(21000),
    gasPrice: web3.utils.toHex(web3.utils.toWei('10', 'gwei'))
  }

  // Sign the transaction
  const tx = new Tx(txObject)
  tx.sign(privateKey1)

  const serializedTx = tx.serialize()
  const raw = '0x' + serializedTx.toString('hex')

  // Broadcast the transaction
  web3.eth.sendSignedTransaction(raw, (err, txHash) => {
    console.log('txHash:', txHash)
    // Now go check etherscan to see the transaction!
  })
})
```

You can run the `app.js` file from your terminal with NodeJS like this:

```
$ node app.js
```

Or simply:

```
$ node app
```

Remember the above also for the other example scripts.

10.9.2 Deploying Smart Contracts

There are multiple ways you can deploy smart contracts to The Ethereum Blockchain. There are even multiple ways to deploy them within Web3.js itself. Like the previous lesson in this series, I'm going to demonstrate one method that will help you better understand what happens when a smart contract is deployed to The Ethereum Blockchain. This example is designed to break the deployment down in to each step in the process.

This lesson will use the same `app.js` file that we used in the previous lesson. We'll set it up like this: Check out this code to follow along with the tutorial:

```
var Tx = require('ethereumjs-tx')
const Web3 = require('web3')
const web3 = new Web3('https://ropsten.infura.io/YOUR_INFURA_API_KEY')

const account1 = '' // Your account address 1

const privateKey1 = Buffer.from('YOUR_PRIVATE_KEY_1', 'hex')
```

This lesson example will consist of the same three basic steps as the previous lesson:

1. Build a transaction object
2. Sign the transaction
3. Send the transaction

These steps are the same because anytime we write data to the blockchain, it always consists of these same basic steps. I'm trying to show you that deploying a smart contact actually looks a lot like sending Ether from one account to another, or calling a smart contract function. We're still building a transaction and sending it to the network. The only difference is the transaction parameters.

Let's go ahead and build the transaction object like this:

```
const txObject = {
  nonce:    web3.utils.toHex(txCount),
  gasLimit: web3.utils.toHex(1000000), // Raise the gas limit to a much higher amount
  gasPrice: web3.utils.toHex(web3.utils.toWei('10', 'gwei')),
  data: data
}
```

We're building this transaction object that has many of the same fields as the object from the previous lesson like `nonce`, `gasLimit`, and `gasPrice`. There are also some key differences. Let's break down each of these:

- `nonce` - this is the previous transaction count for the given account. This is the same a the previous lesson.
- `gasLimit` - this is the maximum amount of gas consumed by the transaction. We'll raise this limit because deploying smart contracts requires much more gas than sending Ether.
- `gasPrice` - this is the amount we want to pay for each unit of gas. This is the same as the previous lesson.
- `value` - this parameter is absent in this example because we aren't sending any Ether in this transaction.

- `to` - this parameter is absent because we aren't sending this transaction to a particular account. Instead we're sending it to the entire network because we're deploying a smart contract!
- `data` - this will be the bytecode of the smart contract that we want to deploy. We'll assign this variable value, and I'll explain this more momentarily.

Let's talk about the `data` parameter. This is the compiled bytecode representation of the smart contract in hexidecimal. In order to obtain this value, we first need a smart contract, and then we need to compile it! You are welcome to use any smart contract you like, especially since we're deploying this to a test network. However, I'm going to use an ERC-20 token smart contract that I built in this video. You can follow along with me in the accompanying Web3.js tutorial video above to watch me compile this particular ERC-20 smart contract with Remix to obtain this data string. Once you've compiled your contract, you can assign the data value to a variable like this:

```
const data =
'0x60806040526040805190810160405280600a81526020017f4441707020546f6b656e000000000000000
000000000000000000000000000000008152506000908051906020019061004f92919061014e565b506040805
190810160405280600481526020017f4441500000000000000000000000000000000000000000000000000000
0000000008152506001908051906020019061009b92919061014e565b506040805190810160405280600f8
1526020017f4441707020546f6b656e2076312e300000000000000000000000000000000000081525060029
0805190602001906100e792919061014e565b503480156100f457600080fd5b506000620f4240905080600
460003373ffffffffffffffffffffffffffffffffffffffff1673fffffffffffffffffffffffffffffffff
fffffff1681526020019081526020016000208190555080600381905550506101f3565b828054600181600
116156101000203166002900490600052602060002090601f016020900481019282601f1061018f5780516
0ff19168380011785556101bd565b82800160010185558215610 1bd579182015b82811115610 1bc5782518
2559160200191906001019061 01a1565b5b5090506101ca91906101ce565b5090565b6101f091905b80821
1156101ec5760008160009055506001016101d4565b5090565b90565b610b998061020 26000396000f3006
08060405260043610610099576000357c01000000000000000000000000000000000000000000000000000
00000900463ffffffff16806306fdde031461009e578063095ea7b31461012e57806318160ddd146101935
7806323b872dd146101be5780635a3b7e42146102435 7806370a08231146102d357806395d89b411461032
a578063a9059cbb146103ba578063dd62ed3e1461041f575b600080fd5b3480156100aa57600080fd5b506
100b3610496565b60405180806020018281038252838181518152602001915080519060200190808383600
05b838110156100f3578082015181840152602081019050610 0d8565b50505050905090810190601f16801
5610120578082038051600183602003610100 0a031916815260200191505b509250505060405180910390f
35b34801561013a57600080fd5b5061017960048036038101908080357 3fffffffffffffffffffffffffff
fffffffffffff1690602001909291908035906020019092919050505061053456 5b6040518082151515158
15260200191505060405180910390f35b34801561019f57600080fd5b506101a8610626565b60405180828
15260200191505060405180910390f35b3480156101ca57600080fd5b50610229600480360381019080803
573ffffffffffffffffffffffffffffffffffffffff16906020019092919080357 3fffffffffffffffffff
fffffffffffffffffffff1690602001909291908035906020019092919050505061062c565b60405180821
51515158152602001915050604051809103 90f35b34801561024f57600080fd5b506102586108 9b565b604
0518080602001828103825283818151815260200191508051906020019080838360005b838110156102985
78082015181840152602081019050610 27d565b50505050905090810190601f1680156102c557808203805
16001836020036101000a031916815260200191505b509250505060405180910390f35b3480156102df576
00080fd5b50610314600480360381019080803573ffffffffffffffffffffffffffffffffffffffff16906
0200190929190505050610939565b60405180828152602001915050604051809103 90f35b3480156103365
7600080fd5b5061033f610951565b604051808060200182810382528381815181526020019150805190602
001908083836000 5b8381101561037f578082015181840152602081019050610364565b50505050905090 8
10190601f1680156103ac5780820380516001836020036101000a031916815260200191505b50925050506
```

```
0405180910390f35b3480156103c657600080fd5b50610405600480360381019080803573fffffffffffff
fffffffffffffffffffffffffff169060200190929190803590602001909291905050506109ef565b60405
1808215151515815260200191505060405180910390f35b34801561042b57600080fd5b506104806004803
60381019080803573ffffffffffffffffffffffffffffffffffffffff169060200190929190803573fffff
fffffffffffffffffffffffffffffffffff169060200190929190505050610b48565b60405180828152602
00191505060405180910390f35b60008054600181600116156101000203166002900480601f01602080910
40260200160405190810160405280929190818152602001828054600181600116156101000203166002900
4801561052c5780601f106105015761010080835404028352916020019161052c565b82019190600052602
0600020905b81548152906001019060200180831161050f57829003601f168201915b505050505081565b6
00081600560003373ffffffffffffffffffffffffffffffffffffffff1673fffffffffffffffffffffffff
fffffffffffffff16815260200190815260200160002060008573fffffffffffffffffffffffffffffffff
fffffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260200160002081905
5508273ffffffffffffffffffffffffffffffffffffffff163373fffffffffffffffffffffffffffffffff
fffffff167f8c5be1e5ebec7d5bd14f71427d1e84f3dd0314c0f7b2291e5b200ac8c7c3b92584604051808
2815260200191505060405180910390a36001905092915050565b60035481565b6000600460008573fffff
fffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff1681526
0200190815260200160002054821115151561067c57600080fd5b600560008573fffffffffffffffffffff
fffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260200
160002060003373ffffffffffffffffffffffffffffffffffffffff1673fffffffffffffffffffffffffff
fffffffffffff16815260200190815260200160002054821115151561070757600080fd5b8160046000867
3ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff1
681526020019081526020016000206000828254039250508190555081600460008573fffffffffffffffff
fffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff1681526020019081526
020016000206000828254019250508190555081600560008673fffffffffffffffffffffffffffffffffff
fffff1673ffffffffffffffffffffffffffffffffffffffff1681526020019081526020016000206000337
3ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff1
68152602001908152602001600020600082825403925050819055508273fffffffffffffffffffffffffff
fffffffffffff168473ffffffffffffffffffffffffffffffffffffffff167fddf252ad1be2c89b69c2b06
8fc378daa952ba7f163c4a11628f55a4df523b3ef846040518082815260200191505060405180910390a36
00190509392505050565b60028054600181600116156101000203166002900480601f01602080910402602
00160405190810160405280929190818152602001828054600181600116156101000203166002900480156
109315780601f1061090657610100808354040283529160200191610931565b82019190600052602060002
0905b81548152906001019060200180831161091457829003601f168201915b505050505081565b6004602
0528060005260406000206000915090505481565b600180546001816001161561010002031660029004806
01f01602080910402602001604051908101604052809291908181526020018280546001816001161561010
00203166002900480156109e75780601f106109bc576101008083540402835291602001916109e7565b820
191906000526020600020905b8154815290600101906020018083116109ca57829003601f168201915b505
050505081565b600081600460003373ffffffffffffffffffffffffffffffffffffffff1673fffffffffff
fffffffffffffffffffffffffffff1681526020019081526020016000205410151515610a3f57600080fd5
b81600460003373ffffffffffffffffffffffffffffffffffffffff1673fffffffffffffffffffffffffff
fffffffffffff1681526020019081526020016000206000828254039250508190555081600460008573fff
fffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff16815
2602001908152602001600020600082825401925050819055508273fffffffffffffffffffffffffffffff
fffffffff163373ffffffffffffffffffffffffffffffffffffffff167fddf252ad1be2c89b69c2b068fc3
78daa952ba7f163c4a11628f55a4df523b3ef846040518082815260200191505060405180910390a360019
05092915050565b60056020528160005260406000206020528060005260406000206000915091505054815
600a165627a7a723058204c3f690997294d337edc3571d8e77afc5b0e56a2f4bfae6fb59139c8e4eb2f7e0
029'
```

Now we can also assign the `nonce` value by getting the transaction count, just like the previous lesson:

```
web3.eth.getTransactionCount(account1, (err, txCount) => {
  const data = '' // Your data value goes here...

  const txObject = {
    nonce:    web3.utils.toHex(txCount),
    gasLimit: web3.utils.toHex(1000000),
    gasPrice: web3.utils.toHex(web3.utils.toWei('10', 'gwei')),
    data: data
  }
})
```

And finally, we can sign this transaction and send it, just like the previous lesson. At this point, the completed tutorial code should look like this:

```
var Tx = require('ethereumjs-tx')
const Web3 = require('web3')
const web3 = new Web3('https://ropsten.infura.io/YOUR_INFURA_API_KEY')

const account1 = '' // Your account address 1
const account2 = '' // Your account address 2

const privateKey1 = Buffer.from('YOUR_PRIVATE_KEY_1', 'hex')
const privateKey2 = Buffer.from('YOUR_PRIVATE_KEY_2', 'hex')

// Deploy the contract
web3.eth.getTransactionCount(account1, (err, txCount) => {
  const                                   data                                   =
'0x608060405260408051908101604052806000a81526020017f4441707020546f6b656e000000000000000
00000000000000000000000000000815250600090805190602001906100 4f92919061014e565b506040805
1908101604052806004815260200 17f44415050000000000000000000000000000000000000000000000000
0000000008152506001908051906020019061009b92919061014e565b506040805190810160405280600f8
1526020017f4441707020546f6b656e2076312e300000000000000000000000000000000000815250600 29
08051906020019061 00e792919061014e565b503480156100f457600080fd5b506000620f4240905080600
460003373ffffffffffffffffffffffffffffffffffffffff1673fffffffffffffffffffffffffffffffff
fffffff1681526020019081526020016000208190555080600381905550506101f3565b82805460018160
0116156101000203166002900490600052602060002090601f016020900481019282601f1061018f5780516
0ff19168380011785556101bd565b828001600101855582156101bd579182015b828111156101bc5782518
25591602001919060010190610 1a1565b5b5090506101ca91906101ce565b5090565b6101f091905b80821
1156101ec5760008160009055506001016101d4565b5090565b90565b610b99806102026000396000f3006
080604052600436106100995760003 57c010000000000000000000000000000000000000000000000000000
00000900463ffffffff16806306fdde031461009e578063095ea7b31461012e57806318160ddd146101935
7806323b872dd146101be5780635a3b7e4214610243578063 70a08231146102d357806395d89b411461032
a578063a9059cbb146103ba578063dd62ed3e1461041f575b600080fd5b3480156100aa57600080fd5b506
100b3610496565b60405180806020018281038252838181518152602001915080519060200190808383600
05b83811015610 0f357808201518184015260208101905061 00d8565b50505050905090810190601f16801
561012057808203805160018360200361010 00a031916815260200191505b509250505060405180910390f
35b3480156101 3a57600080fd5b50610179600480360381019080803573fffffffffffffffffffffffffff
fffffffffffff16906020019092919080359060200190929190505050610534565b604051808215151515 8
15260200191505060405180910390f35b3480156101 9f57600080fd5b506101a8610626565b60405180828
15260200191505060405180910390f35b348015610 1ca57600080fd5b50610229600480360381019080803
573ffffffffffffffffffffffffffffffffffffffff16906020019092919080 3573fffffffffffffffffff
fffffffffffffffffffff1690602001909291908035906020019092919050505061062c565b60405180821
515151581526020019150506040518091 0390f35b34801561024f57600080fd5b5061025861089b565b604
0518080602001828103825283818151815260200191508051906020019080838360005b83811015610298 5
7808201518184015260208101905061 027d565b50505050905090810190 601f16801561 02c557808203805
16001836020036101000a031916815260200191505b509250505060405180910390f35b3480156102df576
00080fd5b506103146004803603810190808035 73ffffffffffffffffffffffffffffffffffffffff16906
02001909291905050506109 39565b6040518082815260200191505060405180910390f35b3480156103365
7600080fd5b5061033f610951565b604051808060200182810382528381815181526020019150805190602
001908083836000 5b8381101561037f57808201518184015260208101905061036 4565b505050509050908
10190601f168015610 3ac5780820380516001836020036101000a031916815260200191505b50925050506
0405180910390f35b348015610 3c657600080fd5b5061040560048036038101908080 3573fffffffffffff
fffffffffffffffffffffffffff1690602001909291908035906020019092919050505061 09ef565b60405
180821515151515815260200191505060405180910390f35b34801561042b57600080fd5b506104806004803
603810190808035 73ffffffffffffffffffffffffffffffffffffffff16906020019092919080 3573fffff
fffffffffffffffffffffffffffffffffff16906020019092919050505061 0b48565b60405180828152602
00191505060405180910390f35b6000805460018160011615610100020316600290048 0601f0160208091 0
4026020016040519081016040528092919081815260200182805460018160011615610100020316600290 0
4801561052c5780601f10610501576101008083540402835291602001916105 2c565b82019190600052602
0600020905b81548152906001019060200180831161050f57829003601f168201915b5050505050 81565b6
```

```
00081600560003373ffffffffffffffffffffffffffffffffffffffff1673fffffffffffffffffffffffff
fffffffffffffff16815260200190815260200160002060008573fffffffffffffffffffffffffffffffff
fffffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260200160002081905
5508273ffffffffffffffffffffffffffffffffffffffff163373fffffffffffffffffffffffffffffffff
fffffff167f8c5be1e5ebec7d5bd14f71427d1e84f3dd0314c0f7b2291e5b200ac8c7c3b92584604051808
2815260200191505060405180910390a36001905092915050565b60035481565b6000600460008573fffff
fffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff1681526
0200190815260200160002054821115151561067c57600080fd5b600560008573fffffffffffffffffffff
fffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260200
160002060003373ffffffffffffffffffffffffffffffffffffffff1673fffffffffffffffffffffffffff
fffffffffffff16815260200190815260200160002054821115151561070757600080fd5b8160046000867
3ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff1
681526020019081526020016000206000828254039250508190555081600460008573fffffffffffffffff
fffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff1681526020019081526
020016000206000828254019250508190555081600560008673fffffffffffffffffffffffffffffffffff
fffff1673ffffffffffffffffffffffffffffffffffffffff1681526020019081526020016000206000337
3ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff1
68152602001908152602001600020600082825403925050819055508273fffffffffffffffffffffffffff
fffffffffffff168473ffffffffffffffffffffffffffffffffffffffff167fddf252ad1be2c89b69c2b06
8fc378daa952ba7f163c4a11628f55a4df523b3ef846040518082815260200191505060405180910390a36
00190509392505050565b6002805460018160011615610100020316600290048060 1f0160208091040260 2
```

```
00190509392505050565b600280546001816001161561010002031660029004806 01f01602080910402602
0016040519081016040528092919081815260200182805460018160011615610100020316600290048 0156
109315780601f10610906576101008083540402835291602001916109315 65b8201919060005260206000 2
0905b81548152906001019060200180831161091457829003601f168201915b505050505081565b6004602
052806000526040600020600091509050548 1565b600180546001816001161561010002031660029004806
01f01602080910402602001604051908101604052809291908181526020018280546001816001161561010
0020316600290048015610 9e75780601f106109bc57610100808354040283529160200191610 9e7565b820
19190600052602060002090 5b81548152906001019060200180831161 09ca57829003601f168201915b505
050505081565b600081600460003373ffffffffffffffffffffffffffffffffffffffff1673fffffffffff
fffffffffffffffffffffffffffff1681526020019081526020016000205410151515610a3f57600080fd5
b81600460003373ffffffffffffffffffffffffffffffffffffffff1673fffffffffffffffffffffffffff
fffffffffffff1681526020019081526020016000206000828254039250508190555081600460008573fff
fffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff16815
26020019081526020016000206000828254019250508190555082 73fffffffffffffffffffffffffffffff
fffffffff163373ffffffffffffffffffffffffffffffffffffffff167fddf252ad1be2c89b69c2b068fc3
78daa952ba7f163c4a11628f55a4df523b3ef846040518082815260200191505060405180910390a360019
05092915050565b6005602052816000526040600020602052806000526040600020600091509150505 4815
600a165627a7a723058204c3f690997294d337edc3571d8e77afc5b0e56a2f4bfae6fb59139c8e4eb2f7e0
029'

  const txObject = {
    nonce:    web3.utils.toHex(txCount),
    gasLimit: web3.utils.toHex(1000000), // Raise the gas limit to a much higher amount
    gasPrice: web3.utils.toHex(web3.utils.toWei('10', 'gwei')),
    data: data
  }

  const tx = new Tx(txObject)
  tx.sign(privateKey1)

  const serializedTx = tx.serialize()
  const raw = '0x' + serializedTx.toString('hex')

  web3.eth.sendSignedTransaction(raw, (err, txHash) => {
    console.log('err:', err, 'txHash:', txHash)
    // Use this txHash to find the contract on Etherscan!
  })
})
```

10.9.3 Calling Smart Contract Functions with Web3.js

This lesson will use many of the same basic tutorial steps as the previous lessons because, like the previous lessons, it's designed to show you all the basic steps required when creating transactions on The Ethereum Blockchain. We'll use the same basic setup with an `app.js` file that will look like this:

```
const Web3 = require('web3')
const web3 = new Web3('https://ropsten.infura.io/YOUR_INFURA_API_KEY')

const account1 = '' // Your account address 1
const account2 = '' // Your account address 2

const privateKey1 = Buffer.from('YOUR_PRIVATE_KEY_1', 'hex')
const privateKey2 = Buffer.from('YOUR_PRIVATE_KEY_2', 'hex')
```

We'll also build out a transaction object, just like this:

```
const txObject = {
  nonce:    web3.utils.toHex(txCount),
  gasLimit: web3.utils.toHex(800000),
  gasPrice: web3.utils.toHex(web3.utils.toWei('10', 'gwei')),
  to: contractAddress,
  data: data
}
```

If you've been following along with the previous lessons, many of these values should look familiar to you. Let's make a note of some changes.

- `to` - this parameter will be the address of the deployed contract. We'll obtain that value and assign it momentarily.
- `data` - this will be the hexidecimal representation of the function we want to call on the smart contract. We'll also assign this value momentarily.

In order to fill these values out, we'll need to get the smart contract ABI for this ERC-20 token. You can follow along with me in the video above as I obtain the ABI from Remix. I'll also need to get the smart contract address from Etherscan (this was available whenever we deployed the smart contract in the last lesson). Now that we have both of these things, we can create a JavaScript representation of the smart contract with Web3.js like this:

```
const contractAddress = '0xd03696B53924972b9903eB17Ac5033928Be7D3Bc'
const contractABI =
[{"constant":true,"inputs":[],"name":"name","outputs":[{"name":"","type":"string"}],"p
ayable":false,"stateMutability":"view","type":"function"},{"constant":false,"inputs":[
{"name":"_spender","type":"address"},{"name":"_value","type":"uint256"}],"name":"appro
ve","outputs":[{"name":"success","type":"bool"}],"payable":false,"stateMutability":"no
npayable","type":"function"},{"constant":true,"inputs":[],"name":"totalSupply","output
s":[{"name":"","type":"uint256"}],"payable":false,"stateMutability":"view","type":"fun
ction"},{"constant":false,"inputs":[{"name":"_from","type":"address"},{"name":"_to","t
ype":"address"},{"name":"_value","type":"uint256"}],"name":"transferFrom","outputs":[{
"name":"success","type":"bool"}],"payable":false,"stateMutability":"nonpayable","type"
:"function"},{"constant":true,"inputs":[],"name":"standard","outputs":[{"name":"","typ
e":"string"}],"payable":false,"stateMutability":"view","type":"function"},{"constant":
true,"inputs":[{"name":"","type":"address"}],"name":"balanceOf","outputs":[{"name":"",
"type":"uint256"}],"payable":false,"stateMutability":"view","type":"function"},{"const
ant":true,"inputs":[],"name":"symbol","outputs":[{"name":"","type":"string"}],"payable
":false,"stateMutability":"view","type":"function"},{"constant":false,"inputs":[{"name
":"_to","type":"address"},{"name":"_value","type":"uint256"}],"name":"transfer","outpu
ts":[{"name":"success","type":"bool"}],"payable":false,"stateMutability":"nonpayable",
"type":"function"},{"constant":true,"inputs":[{"name":"","type":"address"},{"name":"",
"type":"address"}],"name":"allowance","outputs":[{"name":"","type":"uint256"}],"payabl
e":false,"stateMutability":"view","type":"function"},{"inputs":[],"payable":false,"stateMutability":"nonpayable","type":"constructor"},{"anonymous":false,"inputs":[{"indexed":true,"name":"_from","type":"address"},{"indexed":true,"name":"_to","type":"address"},{"indexed":false,"name":"_value","type":"uint256"}],"name":"Transfer","type":"event"},{"anonymous":false,"inputs":[{"indexed":true,"name":"_owner","type":"address"},{"indexed":true,"name":"_spender","type":"address"},{"indexed":false,"name":"_value","type":"uint256"}],"name":"Approval","type":"event"}]

const contract = new web3.eth.Contract(abi, contractAddress)
```

Great! Now we have a JavaScript representation of the deployed contract. Now we can fill out the `data` field of the transaction by converting the contract's `transfer()` function to bytecode (that's the function we'll call on this smart contract). We can do this with the Web3.js function `encodeABI()` that is available on the `contract` object. That looks like this:

```
const data = contract.methods.transfer(account2, 1000).encodeABI()
```

That's it! That's how easy it is to encode this function call for the transaction! Note that we're transferring 1,000 tokens to `account2`. This method takes care of encoding these function parameters for us, too!

Now that's everything we need to build the transaction object. Just like the previous lessons, we can now sign this transaction and send it. Once we do, we can log the values of the account balances to see that the smart contract function was called, and that the token transfers were complete. The complete tutorial code will look like this:

```
const Web3 = require('web3')
const web3 = new Web3('https://ropsten.infura.io/YOUR_INFURA_API_KEY')

const account1 = '' // Your account address 1
const account2 = '' // Your account address 2

const privateKey1 = Buffer.from('YOUR_PRIVATE_KEY_1', 'hex')
const privateKey2 = Buffer.from('YOUR_PRIVATE_KEY_2', 'hex')

// Read the deployed contract - get the addresss from Etherscan
const contractAddress = '0xd03696B53924972b9903eB17Ac5033928Be7D3Bc'
const contractABI =
[{"constant":true,"inputs":[],"name":"name","outputs":[{"name":"","type":"string"}],"payable":false,"stateMutability":"view","type":"function"},{"constant":false,"inputs":[{"name":"_spender","type":"address"},{"name":"_value","type":"uint256"}],"name":"approve","outputs":[{"name":"success","type":"bool"}],"payable":false,"stateMutability":"nonpayable","type":"function"},{"constant":true,"inputs":[],"name":"totalSupply","outputs":[{"name":"","type":"uint256"}],"payable":false,"stateMutability":"view","type":"function"},{"constant":false,"inputs":[{"name":"_from","type":"address"},{"name":"_to","type":"address"},{"name":"_value","type":"uint256"}],"name":"transferFrom","outputs":[{"name":"success","type":"bool"}],"payable":false,"stateMutability":"nonpayable","type":"function"},{"constant":true,"inputs":[],"name":"standard","outputs":[{"name":"","type":"string"}],"payable":false,"stateMutability":"view","type":"function"},{"constant":true,"inputs":[{"name":"","type":"address"}],"name":"balanceOf","outputs":[{"name":"","type":"uint256"}],"payable":false,"stateMutability":"view","type":"function"},{"constant":true,"inputs":[],"name":"symbol","outputs":[{"name":"","type":"string"}],"payable":false,"stateMutability":"view","type":"function"},{"constant":false,"inputs":[{"name":"_to","type":"address"},{"name":"_value","type":"uint256"}],"name":"transfer","outputs":[{"name":"success","type":"bool"}],"payable":false,"stateMutability":"nonpayable","type":"function"},{"constant":true,"inputs":[{"name":"","type":"address"},{"name":"","type":"address"}],"name":"allowance","outputs":[{"name":"","type":"uint256"}],"payable":false,"stateMutability":"view","type":"function"},{"inputs":[],"payable":false,"stateMutability":"nonpayable","type":"constructor"},{"anonymous":false,"inputs":[{"indexed":true,"name":"_from","type":"address"},{"indexed":true,"name":"_to","type":"address"},{"indexed":false,"name":"_value","type":"uint256"}],"name":"Transfer","type":"event"
},{"anonymous":false,"inputs":[{"indexed":true,"name":"_owner","type":"address"},{"ind
exed":true,"name":"_spender","type":"address"},{"indexed":false,"name":"_value","type"
:"uint256"}],"name":"Approval","type":"event"}]

const contract = new web3.eth.Contract(abi, contractAddress)

// Transfer some tokens
web3.eth.getTransactionCount(account1, (err, txCount) => {

  const txObject = {
    nonce:    web3.utils.toHex(txCount),
    gasLimit: web3.utils.toHex(800000), // Raise the gas limit to a much higher amount
    gasPrice: web3.utils.toHex(web3.utils.toWei('10', 'gwei')),
    to: contractAddress,
    data: contract.methods.transfer(account2, 1000).encodeABI()
  }

  const tx = new Tx(txObject)
  tx.sign(privateKey1)

  const serializedTx = tx.serialize()
  const raw = '0x' + serializedTx.toString('hex')

  web3.eth.sendSignedTransaction(raw, (err, txHash) => {
    console.log('err:', err, 'txHash:', txHash)
    // Use this txHash to find the contract on Etherscan!
  })
})

// Check Token balance for account1
contract.methods.balanceOf(account1).call((err, balance) => {
  console.log({ err, balance })
})

// Check Token balance for account2
contract.methods.balanceOf(account2).call((err, balance) => {
  console.log({ err, balance })
})
```

10.9.4 Smart Contract Events with Web3.js

Ethereum smart contracts have the ability to emit events that indicate that something happened within the smart contract code execution. Consumers have the ability to subscribe to these events, and Web3.js will provide us with this functionality. That's exactly what we'll cover in this lesson.

We're going to continue using an ERC-20 smart contract as the reference point for this tutorial because this standard specifies that the smart contract must emit a `Transfer` event anytime an ERC-20 token is transferred. We'll actually connect to the Ethereum main net to subscribe to the `Transfer` event for the OmiseGo ERC-20 token.

Let's go ahead and set up the `app.js` file much like we did in the previous lessons. This time, we'll connect to the Ethereum main net. I'll go ahead and paste in the OmiseGo smart contract ABI and address, which can be obtained from Etherscan (watch the above video for instructions). Once we have both of these things, we can create a JavaScript representation of the smart contract with Web3.js and assign it to a variable. All of that setup looks like this:

```
const Web3 = require('web3')
const web3 = new Web3('https://mainnet.infura.io/YOUR_INFURA_API_KEY')
```

```
// OMG Token Contract
const abi =
[{"constant":true,"inputs":[],"name":"mintingFinished","outputs":[{"name":"","type":"bool"}],"payable":false,"type":"function"},{"constant":true,"inputs":[],"name":"name","outputs":[{"name":"","type":"string"}],"payable":false,"type":"function"},{"constant":false,"inputs":[{"name":"_spender","type":"address"},{"name":"_value","type":"uint256"}],"name":"approve","outputs":[],"payable":false,"type":"function"},{"constant":true,"inputs":[],"name":"totalSupply","outputs":[{"name":"","type":"uint256"}],"payable":false,"type":"function"},{"constant":false,"inputs":[{"name":"_from","type":"address"},{"name":"_to","type":"address"},{"name":"_value","type":"uint256"}],"name":"transferFrom","outputs":[],"payable":false,"type":"function"},{"constant":true,"inputs":[],"name":"decimals","outputs":[{"name":"","type":"uint256"}],"payable":false,"type":"function"},{"constant":false,"inputs":[],"name":"unpause","outputs":[{"name":"","type":"bool"}],"payable":false,"type":"function"},{"constant":false,"inputs":[{"name":"_to","type":"address"},{"name":"_amount","type":"uint256"}],"name":"mint","outputs":[{"name":"","type":"bool"}],"payable":false,"type":"function"},{"constant":true,"inputs":[],"name":"paused","outputs":[{"name":"","type":"bool"}],"payable":false,"type":"function"},{"constant":true,"inputs":[{"name":"_owner","type":"address"}],"name":"balanceOf","outputs":[{"name":"balance","type":"uint256"}],"payable":false,"type":"function"},{"constant":false,"inputs":[],"name":"finishMinting","outputs":[{"name":"","type":"bool"}],"payable":false,"type":"function"},{"constant":false,"inputs":[],"name":"pause","outputs":[{"name":"","type":"bool"}],"payable":false,"type":"function"},{"constant":true,"inputs":[],"name":"owner","outputs":[{"name":"","type":"address"}],"payable":false,"type":"function"},{"constant":true,"inputs":[],"name":"symbol","outputs":[{"name":"","type":"string"}],"payable":false,"type":"function"},{"constant":false,"inputs":[{"name":"_to","type":"address"},{"name":"_value","type":"uint256"}],"name":"transfer","outputs":[],"payable":false,"type":"function"},{"constant":false,"inputs":[{"name":"_to","type":"address"},{"name":"_amount","type":"uint256"},{"name":"_releaseTime","type":"uint256"}],"name":"mintTimelocked","outputs":[{"name":"","type":"address"}],"payable":false,"type":"function"},{"constant":true,"inputs":[{"name":"_owner","type":"address"},{"name":"_spender","type":"address"}],"name":"allowance","outputs":[{"name":"remaining","type":"uint256"}],"payable":false,"type":"function"},{"constant":false,"inputs":[{"name":"newOwner","type":"address"}],"name":"transferOwnership","outputs":[],"payable":false,"type":"function"},{"anonymous":false,"inputs":[{"indexed":true,"name":"to","type":"address"},{"indexed":false,"name":"value","type":"uint256"}],"name":"Mint","type":"event"},{"anonymous":false,"inputs":[],"name":"MintFinished","type":"event"},{"anonymous":false,"inputs":[],"name":"Pause","type":"event"},{"anonymous":false,"inputs":[],"name":"Unpause","type":"event"},{"anonymous":false,"inputs":[{"indexed":true,"name":"owner","type":"address"},{"indexed":true,"name":"spender","type":"address"},{"indexed":false,"name":"value","type":"uint256"}],"name":"Approval","type":"event"},{"anonymous":false,"inputs":[{"indexed":true,"name":"from","type":"address"},{"indexed":true,"name":"to","type":"address"},{"indexed":false,"name":"value","type":"uint256"}],"name":"Transfer","type":"event"}]
const address = '0xd26114cd6EE289AccF82350c8d8487fedB8A0C07'

const contract = new web3.eth.Contract(abi, address)
```

Now we can look at the past events for this smart contract with the getPastEvents() function available on our contract object. First, let's get all of the events emitted by the contract, for its entire lifetime:

```
contract.getPastEvents(
  'AllEvents',
  {
    fromBlock: 0,
    toBlock: 'latest'
  },
  (err, events) => { console.log(events) }
)
```

Here, this function takes two arguments: the event name, and a set of filtering parameters. We specify that we want to listen to all events by passing `'AllEvents'`. We'll specify a specific event momentarily. Then, we pass some filtering parameters that specify that we want to get events for the entire lifetime of this contract by passing `from: 0`, or the first block in the chain, to `toBlock: 'latest'`, or the latest block in the chain. Just a note, if you run this code, it will probably fail execution because the event stream is so large for this particular contract on the Ethereum main net!

Let's aim for a successful execution by limiting the number of blocks we want to stream from. We can pass in a more recent `fromBlock` like this:

```
contract.getPastEvents(
  'AllEvents',
  {
    fromBlock: 5854000,
    toBlock: 'latest'
  },
  (err, events) => { console.log(events) }
)
```

Ah, that's much better. Now, we can also specify that we *just* want to listen to the `Transfer` event like this:

```
contract.getPastEvents(
  'Transfer',
  {
    fromBlock: 5854000,
    toBlock: 'latest'
  },
  (err, events) => { console.log(events) }
)
```

And that's it! That's all the code you need to see all of the recent transfer events for the OmiseGo ERC-20 token. With this code, you could easily build something like a transaction history for the OMG token in a crypto wallet. That's the power of Web3.js. At this point, the completed tutorial code should look like this:

```
const Web3 = require('web3')
const web3 = new Web3('https://mainnet.infura.io/YOUR_INFURA_API_KEY')

// OMG Token Contract
const abi =
[{"constant":true,"inputs":[],"name":"mintingFinished","outputs":[{"name":"","type":"b
ool"}],"payable":false,"type":"function"},{"constant":true,"inputs":[],"name":"name","
outputs":[{"name":"","type":"string"}],"payable":false,"type":"function"},{"constant":
false,"inputs":[{"name":"_spender","type":"address"},{"name":"_value","type":"uint256"
}],"name":"approve","outputs":[],"payable":false,"type":"function"},{"constant":true,"
inputs":[],"name":"totalSupply","outputs":[{"name":"","type":"uint256"}],"payable":fal
se,"type":"function"},{"constant":false,"inputs":[{"name":"_from","type":"address"},{"
name":"_to","type":"address"},{"name":"_value","type":"uint256"}],"name":"transferFrom
","outputs":[],"payable":false,"type":"function"},{"constant":true,"inputs":[],"name":
"decimals","outputs":[{"name":"","type":"uint256"}],"payable":false,"type":"function"}
,{"constant":false,"inputs":[],"name":"unpause","outputs":[{"name":"","type":"bool"}],
"payable":false,"type":"function"},{"constant":false,"inputs":[{"name":"_to","type":"a
ddress"},{"name":"_amount","type":"uint256"}],"name":"mint","outputs":[{"name":"","typ
e":"bool"}],"payable":false,"type":"function"},{"constant":true,"inputs":[],"name":"pa
used","outputs":[{"name":"","type":"bool"}],"payable":false,"type":"function"},{"const
ant":true,"inputs":[{"name":"_owner","type":"address"}],"name":"balanceOf","outputs":[
{"name":"balance","type":"uint256"}],"payable":false,"type":"function"},{"constant":fa
lse,"inputs":[],"name":"finishMinting","outputs":[{"name":"","type":"bool"}],"payable"
:false,"type":"function"},{"constant":false,"inputs":[],"name":"pause","outputs":[{"na
me":"","type":"bool"}],"payable":false,"type":"function"},{"constant":true,"inputs":[]
,"name":"owner","outputs":[{"name":"","type":"address"}],"payable":false,"type":"funct
ion"},{"constant":true,"inputs":[],"name":"symbol","outputs":[{"name":"","type":"strin
g"}],"payable":false,"type":"function"},{"constant":false,"inputs":[{"name":"_to","typ
e":"address"},{"name":"_value","type":"uint256"}],"name":"transfer","outputs":[],"paya
ble":false,"type":"function"},{"constant":false,"inputs":[{"name":"_to","type":"addres
s"},{"name":"_amount","type":"uint256"},{"name":"_releaseTime","type":"uint256"}],"nam
e":"mintTimelocked","outputs":[{"name":"","type":"address"}],"payable":false,"type":"f
unction"},{"constant":true,"inputs":[{"name":"_owner","type":"address"},{"name":"_spen
der","type":"address"}],"name":"allowance","outputs":[{"name":"remaining","type":"uint
256"}],"payable":false,"type":"function"},{"constant":false,"inputs":[{"name":"newOwne
r","type":"address"}],"name":"transferOwnership","outputs":[],"payable":false,"type":"
function"},{"anonymous":false,"inputs":[{"indexed":true,"name":"to","type":"address"},
{"indexed":false,"name":"value","type":"uint256"}],"name":"Mint","type":"event"},{"ano
nymous":false,"inputs":[],"name":"MintFinished","type":"event"},{"anonymous":false,"in
puts":[],"name":"Pause","type":"event"},{"anonymous":false,"inputs":[],"name":"Unpause
","type":"event"},{"anonymous":false,"inputs":[{"indexed":true,"name":"owner","type":"
address"},{"indexed":true,"name":"spender","type":"address"},{"indexed":false,"name":"
value","type":"uint256"}],"name":"Approval","type":"event"},{"anonymous":false,"inputs
":[{"indexed":true,"name":"from","type":"address"},{"indexed":true,"name":"to","type":
"address"},{"indexed":false,"name":"value","type":"uint256"}],"name":"Transfer","type"
:"event"}]
```

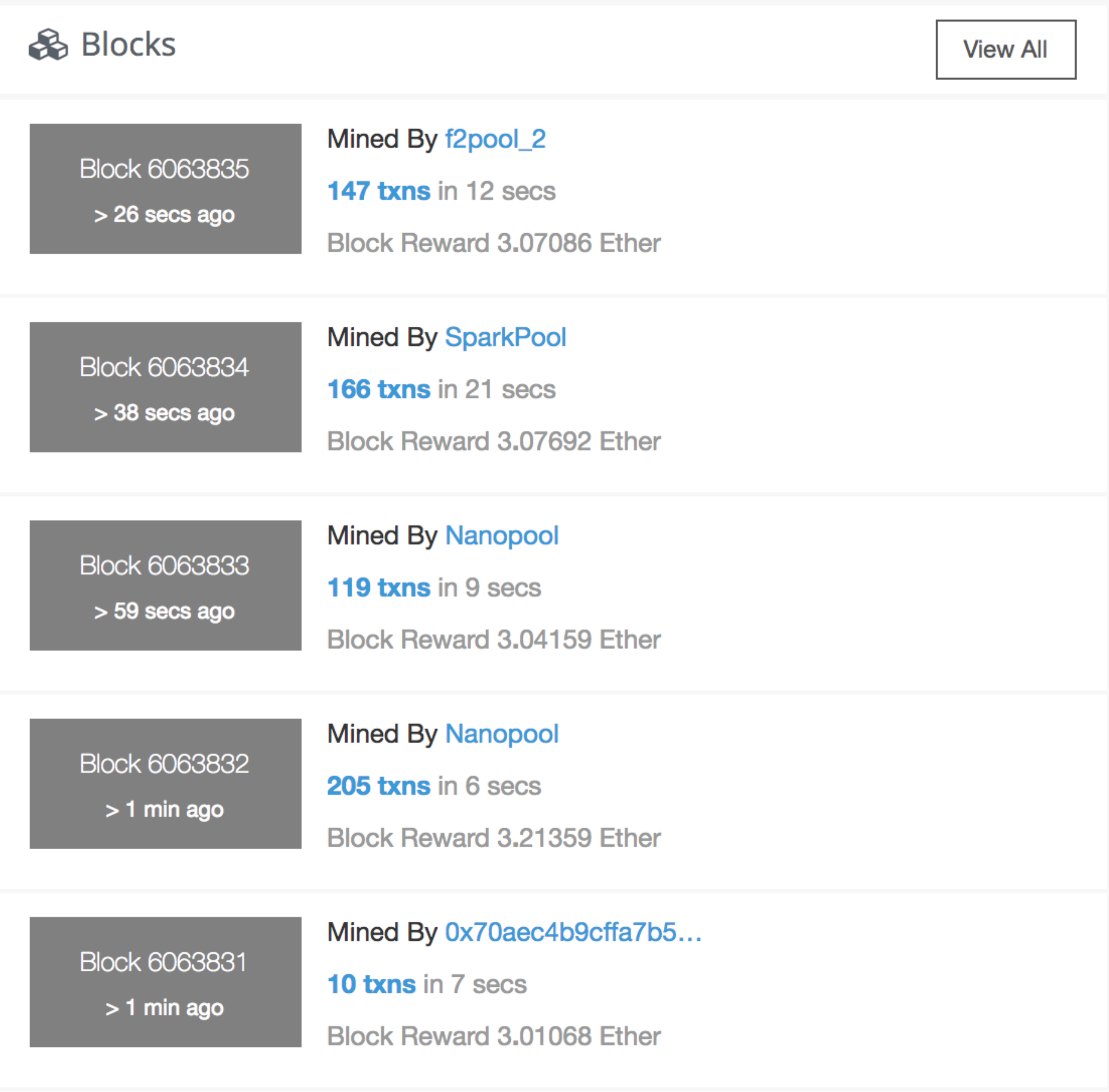
```
const address = '0xd26114cd6EE289AccF82350c8d8487fedB8A0C07'
const contract = new web3.eth.Contract(abi, address)
```

```
// Get Contract Event Stream
contract.getPastEvents(
  'AllEvents',
  {
    fromBlock: 5854000,
    toBlock: 'latest'
  },
  (err, events) => { console.log(events) }
)
```

10.9.5 Inspecting Blocks with Web3.js

This is the seventh video in the 8-part tutorial series. This video will show you how to inspect blocks The Ethereum Blockchain with Web3.js.

Inspecting blocks is often useful when analyzing history on The Ethereum Blockchain. Web3.js has lots of functionality that helps us to do just that. For example, we could build something that looks like this block history feature on Etherscan:



Blocks

View All

Block 6063835
> 26 secs ago

Mined By f2pool_2
147 txns in 12 secs
Block Reward 3.07086 Ether

Block 6063834
> 38 secs ago

Mined By SparkPool
166 txns in 21 secs
Block Reward 3.07692 Ether

Block 6063833
> 59 secs ago

Mined By Nanopool
119 txns in 9 secs
Block Reward 3.04159 Ether

Block 6063832
> 1 min ago

Mined By Nanopool
205 txns in 6 secs
Block Reward 3.21359 Ether

Block 6063831
> 1 min ago

Mined By 0x70aec4b9cffa7b5…
10 txns in 7 secs
Block Reward 3.01068 Ether

Let's set up an `app.js` file to start using some of this functionality provided by Web3.js. This setup will be much simpler than the previous lessons. We'll connect to the main net to inspect blocks there:

```
const Web3 = require('web3')
const web3 = new Web3('https://mainnet.infura.io/YOUR_INFURA_API_KEY')
```

First, we can get the latest block number like this:

```
web3.eth.getBlockNumber().then(console.log)
```

We can also get all the data for the latest block like this:

```
web3.eth.getBlock('latest').then(console.log)
```

You can watch the video above as I explain all the data that gets logged by this function. If we were going to build a block history feature like the one on Etherscan pictured above, we would need to get a list of the most recent blocks in the chain. We can do this by fetching the most recent block and counting backwards until we have the last 10 blocks in the chain. We can do that with a `for` loop like this:

```
web3.eth.getBlockNumber().then((latest) => {
  for (let i = 0; i < 10; i++) {
    web3.eth.getBlock(latest - i).then(console.log)
  }
})
```

Web3.js has another nice feature that allows you to inspect transactions contained within a specific block. We can do that like this:

```
const hash = '0x66b3fd79a49dafe44507763e9b6739aa0810de2c15590ac22b5e2f0a3f502073'
web3.eth.getTransactionFromBlock(hash, 2).then(console.log)
```

The hash is the block’s hash and uniquely identifies it, while the second is the transaction index (in a block there can be even 1000 transactions). That's it! That's how easy it is to inspect blocks with Web3.js. Check out the video above for more in depth explanation of the data returned by the blocks. At this point, all of the tutorial code should look like this:

```
const Web3 = require('web3')
const web3 = new Web3('https://mainnet.infura.io/YOUR_INFURA_API_KEY')

// get latest block number
web3.eth.getBlockNumber().then(console.log)

// // get latest block
web3.eth.getBlock('latest').then(console.log)

// get latest 10 blocks
web3.eth.getBlockNumber().then((latest) => {
  for (let i = 0; i < 10; i++) {
    web3.eth.getBlock(latest - i).then(console.log)
  }
})

// get transaction from specific block
const hash = '0x66b3fd79a49dafe44507763e9b6739aa0810de2c15590ac22b5e2f0a3f502073'
web3.eth.getTransactionFromBlock(hash, 2).then(console.log)
```

10.9.6 Web3.js Utilities

This lesson is designed to show you some cool tips and tricks that you might not know about Web3.js! Let's go ahead and set up the `app.js` and jump into examining these tips. Let's connect to the Ethereum main net like this:

```
const Web3 = require('web3')
const web3 = new Web3('https://mainnet.infura.io/YOUR_INFURA_API_KEY')
```

First, you can actually get the average gas price currently for the network like this:

```
web3.eth.getGasPrice().then((result) => {
  console.log(web3.utils.fromWei(result, 'ether'))
})
```

If you've developed on the blockchain before, you have probably dealt with hashing functions. Web3.js has a lot of built in helpers for using hashing functions. You have direct access to the `sha3` function like this:

```
console.log(web3.utils.sha3('Dapp University'))
```

Or as keccack256:

```
console.log(web3.utils.keccak256('Dapp University'))
```

You can also handle (pseudo) randomness by generating a 32 byte random hex like this:

```
console.log(web3.utils.randomHex(32))
```

Have you ever found yourself trying to perform an action on a JavaScript array or object, and needed the help of an external library? Thankfully, Web3.js ships with the underscoreJS library:

```
const _ = web3.utils._
_.each({ key1: 'value1', key2: 'value2' }, (value, key) => {
  console.log(key)
})
```

And that's it! Those are some fancy tips and tricks you can use with Web3.js. Here is the complete tutorial code for this lesson:

```
const Web3 = require('web3')
const web3 = new Web3('https://mainnet.infura.io/YOUR_INFURA_API_KEY')

// Get average gas price in wei from last few blocks median gas price
web3.eth.getGasPrice().then((result) => {
  console.log(web3.utils.fromWei(result, 'ether')
})

// Use sha256 Hashing function
console.log(web3.utils.sha3('Dapp University'))

// Use keccak256 Hashing function (alias)
console.log(web3.utils.keccak256('Dapp University'))

// Get a Random Hex
console.log(web3.utils.randomHex(32))

// Get access to the underscore JS library
const _ = web3.utils._

_.each({ key1: 'value1', key2: 'value2' }, (value, key) => {
  console.log(key)
})
```

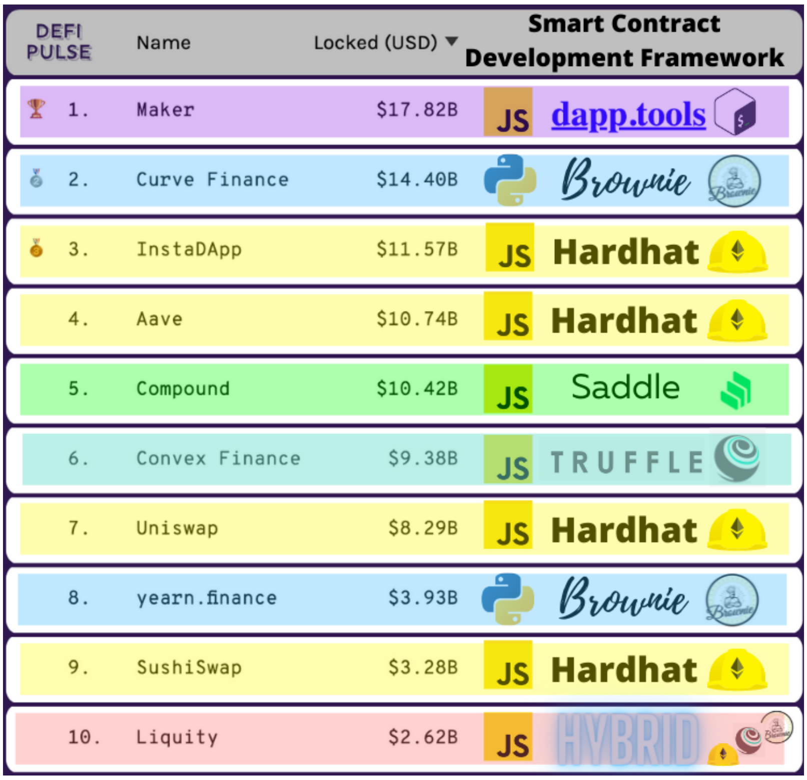
10.10 ether.js

Ethers.js is also an Ethereum JavaScript library that enables developers to communicate and interact with the Ethereum network. Moreover, it is an open-source library with the MIT License. So, what’s the point of Ethers.js if it serves the same purpose as Web3.js? Well, keep in mind that having options is normally a good thing. As such, Ethers.js offers an impressive (in many aspects a superior) alternative to Web3.js. However, just like with any product out there, Ethers.js and Web3.js have their own drawbacks and benefits. More on that in the “Web3.js vs Ethers.js – A Comparison” section below. Find out more here:

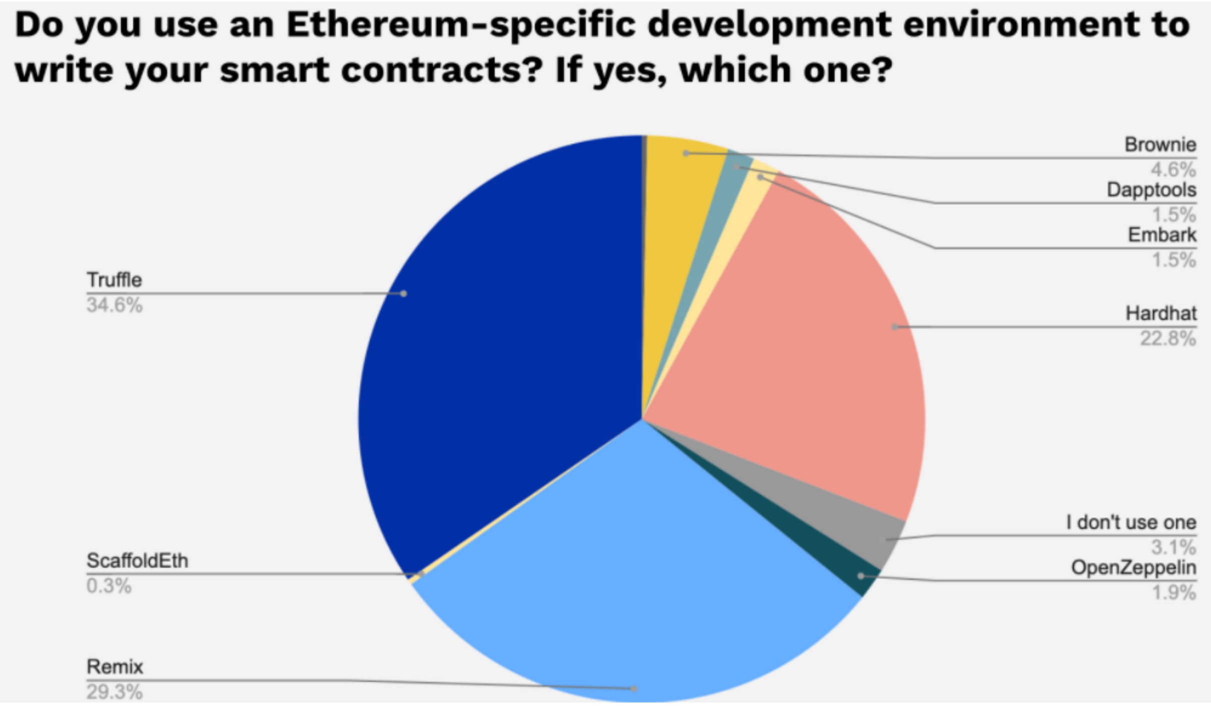
https://moralis.io/web3-js-vs-ethers-js-guide-to-eth-javascript-libraries/

11 Smart contracts development tools

Follows hereafter a list of the development tool used by the highest amount of locked money in **Defi** projects.



DEFI PULSE	Name	Locked (USD) ▼	Smart Contract Development Framework
1.	Maker	$17.82B	JS dapp.tools
2.	Curve Finance	$14.40B	Brownie
3.	InstaDApp	$11.57B	JS Hardhat
4.	Aave	$10.74B	JS Hardhat
5.	Compound	$10.42B	JS Saddle
6.	Convex Finance	$9.38B	JS TRUFFLE
7.	Uniswap	$8.29B	JS Hardhat
8.	yearn.finance	$3.93B	Brownie
9.	SushiSwap	$3.28B	JS Hardhat
10.	Liquity	$2.62B	JS HYBRID



Do you use an Ethereum-specific development environment to write your smart contracts? If yes, which one?

11.1 Web3

It's a Python library built to interact with blockchains, it shouldn't be really considered a development tool because it's required to manually manage quite a lot of stuff, that you don't want to manage when you develop something new. In any case, doing at least once such an exercise is useful to better understand how a blockchain works, what parameters you need to manage to interact with it, what happens 'under the hood' when you use Brownie. There is an “equivalent” (not necessarily all the features are exactly the same) library called Web3.js that is written in Javascript.

```
from solcx import compile_standard, install_solc
import json
from web3 import Web3
import os
from dotenv import load_dotenv

load_dotenv()

# this was not included in youtube video
# solcx.install_solc('0.6.0')

with open("<path to SStorage.sol>/SimpleStorage.sol", "r") as file:
    simple_storage_file = file.read()

compiled_sol = compile_standard(
    {
        "language": "Solidity",
        "sources": {"SimpleStorage.sol": {"content": simple_storage_file}},
        "settings": {
            "outputSelection": {
                "*": {
                    "*": ["abi", "metadata", "evm.bytecode", "evm.sourceMap"]}
            }
        },
    },
    solc_version="0.6.0",
)

# with open("<path to SStorage.sol>/comp_ctrt.json", "w") as file:
#    json.dump(compiled_sol, file)

abi = compiled_sol["contracts"]["SimpleStorage.sol"]["SimpleStorage"]["abi"]
bytecode = compiled_sol["contracts"]["SimpleStorage.sol"]["SimpleStorage"]["evm"]["bytecode"]["object"]

w3 = Web3(Web3.HTTPProvider("http://127.0.0.1:8545"))
# w3 = Web3(Web3.HTTPProvider("http://rinkeby.infura.io/v3/fdjsiaipoflòasdjflkfs/"))

# this is going to be a Ganache local blockchain Network ID ... apparently it’s a
# configurable parameter, but in my opinion it’s not working and it can’t be changed.
# I tried to change it to 5777 but I got a Phyton error that the network id was 1337
chain_id = 1337
my_address = "0x3aa68088CF3387E9771f8d4b4476e77DD420dBb1"

# a private key should NEVER be written in clear, it should be saved in a .env file
# the ".env" file should also be included in the ".gitignore" file, so that it is NOT
# accidentally uploaded to the internet, when synching the projects with github
private_key = os.getenv("PRIVATE_KEY")


SimpleStorage = w3.eth.contract(abi=abi, bytecode=bytecode)

# nonce is a number to avoid ‘reply attacks’, or multiple transactions made with the
# same account that should be considered in order. The nonce can’t never be lower than
# the last one appearing in the blockchain
nonce = w3.eth.getTransactionCount(my_address)

# in Patrick’s video the gasPrice was a missing parameter, but it’s mandatory to have it
transaction = SimpleStorage.constructor().buildTransaction({
    "gasPrice": w3.eth.gas_price,
    "chainId": chain_id,
    "from": my_address,
    "nonce": nonce,
})
```

```
# before doing a transaction on the blockchain, you need to sign it with the account
# that is going to pay the fee for it. The message is of course signed with the private
# key of the account, without revealing the private key.
signed_txn = w3.eth.account.sign_transaction(
    transaction, private_key=private_key)
tx_hash = w3.eth.send_raw_transaction(signed_txn.rawTransaction)
tx_receipt = w3.eth.wait_for_transaction_receipt(tx_hash)
# get the public address of the deployed contract
simple_storage = w3.eth.contract(address=tx_receipt.contractAddress, abi=abi)


# difference between call and transaction
print(simple_storage.functions.retrieve().call())
# this will not work ... if we change something on the blockchain, we need to
# create a transaction, sign it and go on ...
print(simple_storage.functions.store(15).call())
print(simple_storage.functions.retrieve().call())

store_transaction = simple_storage.functions.store(15).buildTransaction({
    "gasPrice": w3.eth.gas_price,
    "chainId": chain_id,
    "from": my_address,
    "nonce": nonce+1})
signed_store_txn = w3.eth.account.sign_transaction(
    store_transaction, private_key=private_key)
send_store_tx = w3.eth.send_raw_transaction(signed_store_txn.rawTransaction)
tx_store_receipt = w3.eth.wait_for_transaction_receipt(send_store_tx)
# this time it's gonna work
print(simple_storage.functions.retrieve().call())
```

11.2 Brownie

It’s a ‘smart contract’ deployment tool, heavily based on ‘Web3.py’.

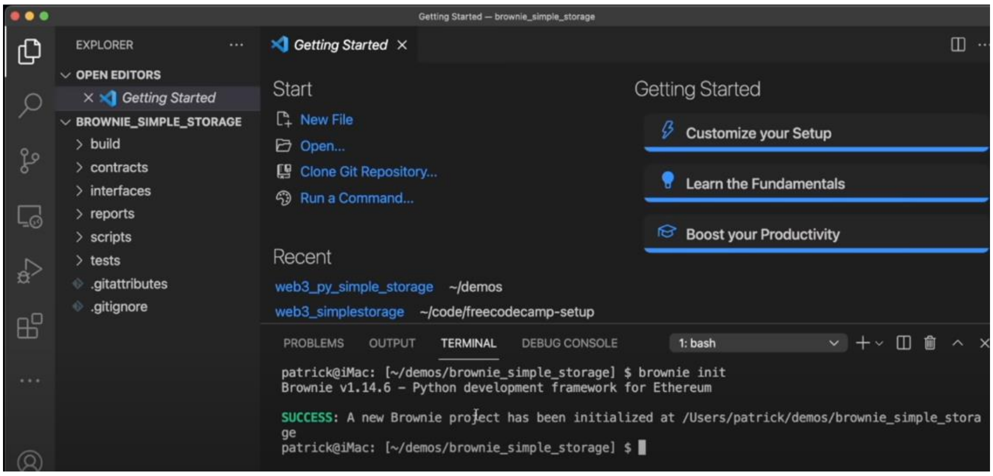
```
code .
```

... opens a new ‘Visual Studio Code’ instance with the selected directory as the working one. It’s NOT used for building GUI or UI, even though Python also has support for building graphical user interfaces (but not so many people use it).

The recommended way to install brownie is through **pipx**, it’s the ‘well known’ pip installer’s brother, but installs everything in a virtual environment, and you don’t need to activate this environment before using brownie. You can find installation details on github’s repository, on Windows:

```
python -m pip install --user pipx
pipx ensurepath
# upgrading pipx
python3 -m pip install --user -U pipx

(close and reopen the powershell terminal)
pipx install eth-brownie

# test successful installation
brownie --version

brownie init
```

... creates a complete tree structure of directories to manage your project:



```
patrick@iMac: [~/demos/brownie_simple_storage] $ brownie init
Brownie v1.14.6 - Python development framework for Ethereum

SUCCESS: A new Brownie project has been initialized at /Users/patrick/demos/brownie_simple_stora
ge
patrick@iMac: [~/demos/brownie_simple_storage] $
```

- build/ : Compiled contracts and test data
- contracts/ : Contract source code
- reports/ : JSON report files for use in the Viewing Coverage Data
- scripts/ : Scripts for **deployment** and interaction
- tests/ : Scripts for testing your project
- brownie-config.yaml : Configuration file for the project

11.2.1 Deploy scripts

```
brownie run /scripts/deploy.py
```

```
Launching 'ganache-cli --port 8545 --gasLimit 12000000 --accounts 10 --hardfork istanbul
--mnemonic brownie
```

Brownie launches every time it runs, a LOCAL ganache instance (i.e. e local Ethereum blockchain), there should not be other instances running locally (or you could get an error:
"Most likely the issue you're dealing with is because ganache is already running in another active project, in order to have brownie recognize ganache is to make sure that's the only environment running ganache close to the project running the node. Which, is most likely the web3 simple storage file... not the newly created brownie file."

```
from dot_env import load_dotenv
```

Brownie has a library to manage accounts of the blockchain that is being used, for example:

```
from brownie import accounts

def deploy_SStorage():
    account = accounts[0]
    print(account)

def main():
    deploy_SStorage()
```

The above will work with a local blockchain, not with a test or an external one.

```
PS C:\Users\<user>\PyScripts\FCC\brownie_SStorage> brownie.exe run .\scripts\deploy.py
INFORMAZIONI: impossibile trovare file corrispondenti ai
criteri di ricerca indicati.
Brownie v1.18.1 - Python development framework for Ethereum
```

```
BrownieSstorageProject is the active project.

Launching  'ganache-cli.cmd  --port  8545  --gasLimit  12000000  --accounts  10  --hardfork
istanbul --mnemonic brownie'...

Running 'scripts\deploy.py::main'...
0x66aB6D9362d4F35596279692F0251Db635165871
Terminating local RPC client...
```

For external blockchains you can create new accounts, adding the private key (not a real one, always a ‘fake’/test one) from the command line:

```
brownie accounts new freecodecamp-account
```

The private key is asked for, you can grab your own from metamask and past it here. Always remember to add ‘0x’ at the beginning (from Metamask usually there is no leading ‘0x’).

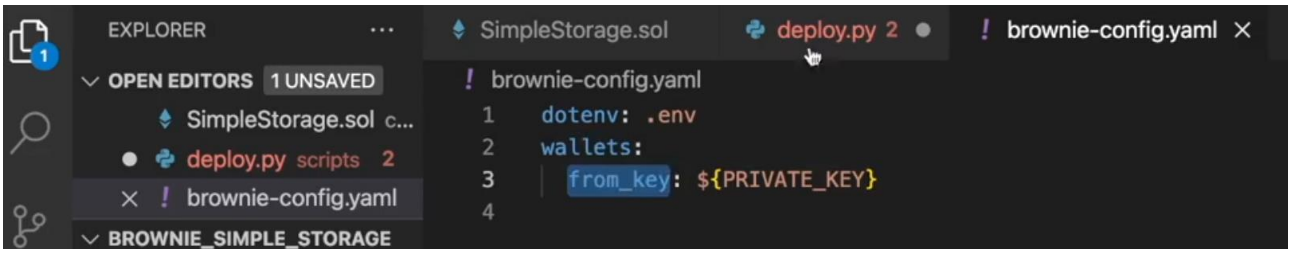
```
brownie accounts list
brownie accounts delete testing
```

```
from brownie import accounts

def deploy_simple_storage():
  account = accounts.load(“freecodecamp-account”)
  print(account)
```

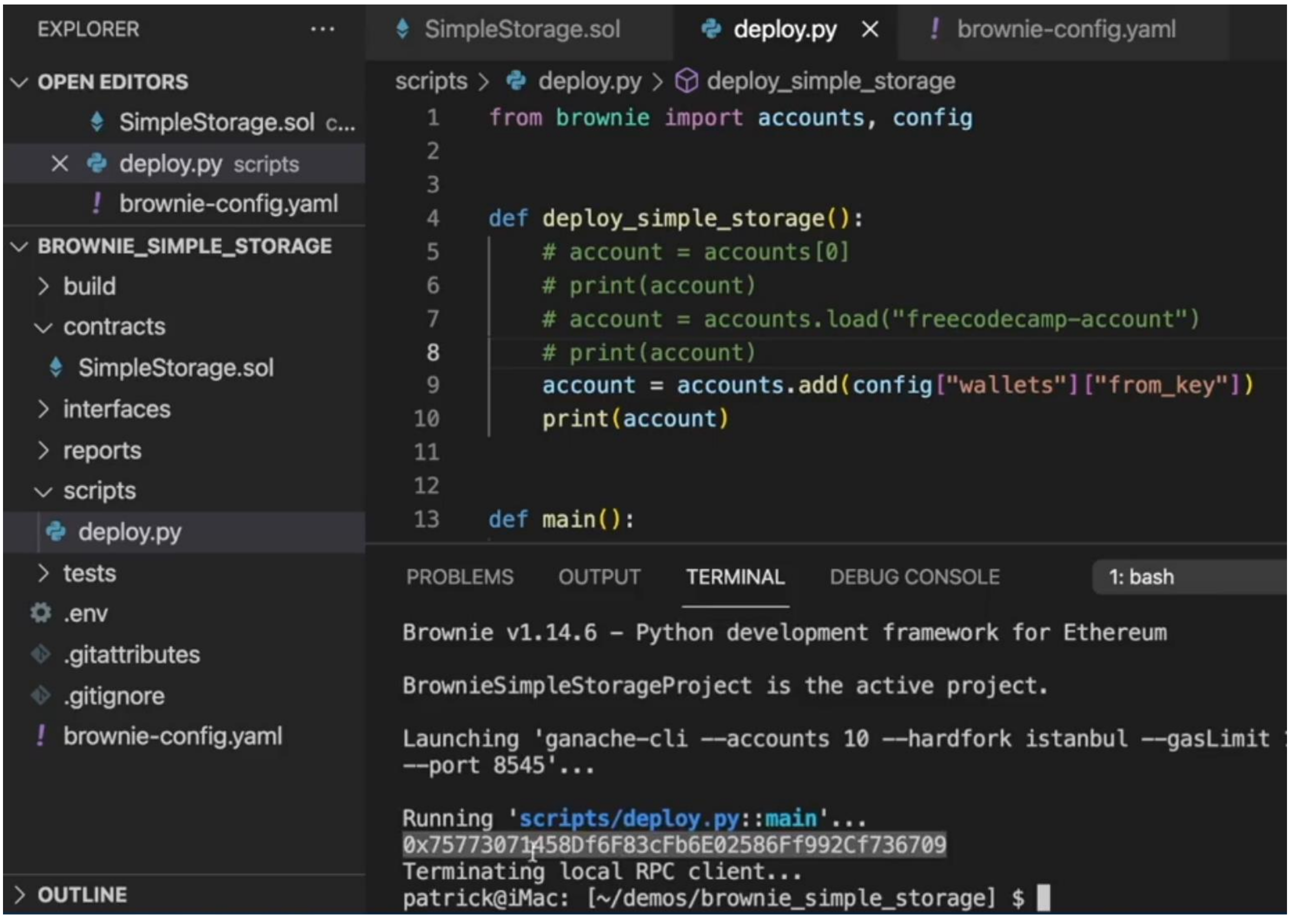
In this way the private-key is password encrypted (it will be asked every time it’s needed) and it should not be possible to upload the password on github in clear text. Moreover, the private kay is also stored locally but in an encrypted way, thus if anyone gains access to your PC, it can’t read your private key. The private-key **could also be saved into a ‘.env’ file to later load it, but it should not be done for real accounts with real money** inside them.
In the main directory of the project, there CAN BE a file called ‘**brownie-config.yaml**’:

```
dotenv: .env
```

In the same directory you can have the ‘.env’ file with the environment variables, that should be loaded in the beginning by brownie. You can use os.getenv(‘variable_name’) to load (for example) the private key, or the “config” dictionary (of dictionaries) as showed in the below example:



```
dotenv: .env
wallets:
  from_key: ${PRIVATE_KEY}
```



```
from brownie import accounts, config


def deploy_simple_storage():
    # account = accounts[0]
    # print(account)
    # account = accounts.load("freecodecamp-account")
    # print(account)
    account = accounts.add(config["wallets"]["from_key"])
    print(account)


def main():
```

```
Brownie v1.14.6 - Python development framework for Ethereum

BrownieSimpleStorageProject is the active project.

Launching 'ganache-cli --accounts 10 --hardfork istanbul --gasLimit
--port 8545'...

Running 'scripts/deploy.py::main'...
0x75773071458Df6F83cFb6E02586Ff992Cf736709
Terminating local RPC client...
patrick@iMac: [~/demos/brownie_simple_storage] $
```

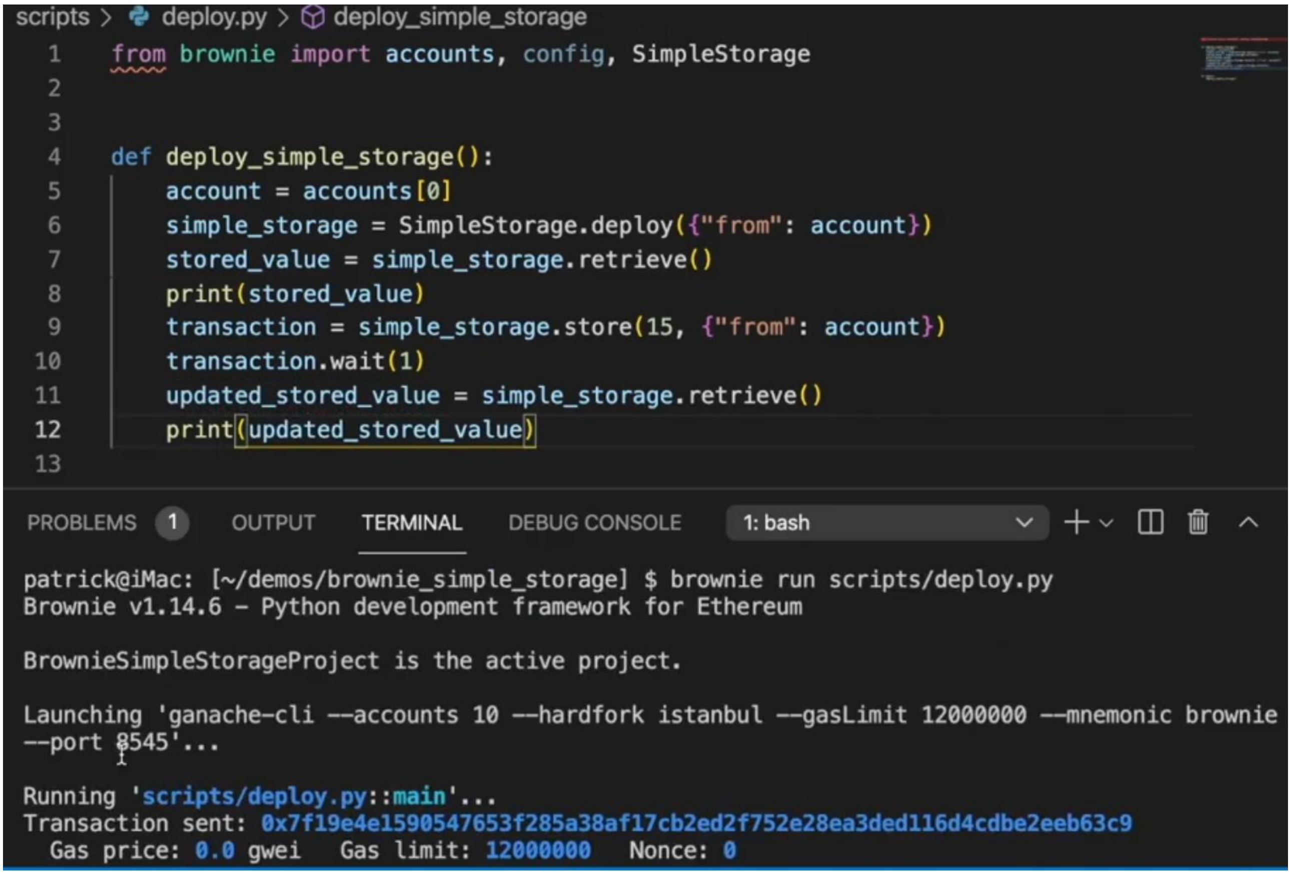
The following deploy.py version:

```
from brownie import accounts, config, SimpleStorage

def deploy_SStorage():
    account = accounts[0]
    SimpleStorage.deploy({"from": account})

def main():
    deploy_SStorage()
```

... does all the job for us:

```
PS C:\<path>\brownie_SStorage> brownie.exe run .\scripts\deploy.py

Brownie v1.18.1 - Python development framework for Ethereum

BrownieSstorageProject is the active project.

Launching 'ganache-cli.cmd --port 8545 --gasLimit 12000000 --accounts 10 --hardfork
istanbul --mnemonic brownie'...

Running 'scripts\deploy.py::main'...
Transaction sent: 0xffe6d801527d91c84ed8711022c296fbb669b6e48f5097241707d76cc6038bfe
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 0
  SimpleStorage.constructor confirmed   Block: 1   Gas used: 335404 (2.80%)
  SimpleStorage deployed at: 0x3194cBDC3dbcd3E11a07892e7bA5c3394048Cc87

Terminating local RPC client...
```

With the above example, you do NOT need to create the transaction data with the Web3 library, get the nonce and many other complex parameters, and you deploy things simply with ONE single command. Of course knowing what happens ‘under the hood’ can make the difference between a ‘smanettone’ and an engineer.



```python
from brownie import accounts, config, SimpleStorage


def deploy_simple_storage():
    account = accounts[0]
    simple_storage = SimpleStorage.deploy({"from": account})
    stored_value = simple_storage.retrieve()
    print(stored_value)
    transaction = simple_storage.store(15, {"from": account})
    transaction.wait(1)
    updated_stored_value = simple_storage.retrieve()
    print(updated_stored_value)

```

```
patrick@iMac: [~/demos/brownie_simple_storage] $ brownie run scripts/deploy.py
Brownie v1.14.6 - Python development framework for Ethereum

BrownieSimpleStorageProject is the active project.

Launching 'ganache-cli --accounts 10 --hardfork istanbul --gasLimit 12000000 --mnemonic brownie
--port 8545'...

Running 'scripts/deploy.py::main'...
Transaction sent: 0x7f19e4e1590547653f285a38af17cb2ed2f752e28ea3ded116d4cdbe2eeb63c9
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 0
```

11.2.2 Test python scripts

Testing is a fundamental part for every software, and it **MUST** be automated. You can’t test everything manually, everything should be automated. Sometimes scripts will need to be updated, but always automated. The following command:

```
brownie test [tests/<test script>] [--interactive]
```

... executes the scripts inside the directory ‘tests’, a specific python script, and optionally if there is a test failure (an assert that is false) you can open a brownie console stopping the exection so that you can query for variables, transactions, and debug what’s going wrong. As usual Brownie documentation is very well done and complete, you can find it here with a lot of useful examples:

https://eth-brownie.readthedocs.io/en/stable/tests-pytest-intro.html#getting-started

```python
from brownie import SimpleStorage, accounts

def test_deploy():
    # arrange
    # act
    # assert
    account = accounts[0]
    sstorage = SimpleStorage.deploy({"from": account})
    value = sstorage.retrieve()
    assert (value == 0)

def test_update():
    account = accounts[0]
    sstorage = SimpleStorage.deploy({"from": account})
    transaction = sstorage.store(15, {"from": account})
    transaction.wait(1)
    assert (sstorage.retrieve() == 15)

brownie test [-k <test function>]
# to debug in case of test failure
brownie test --pdb
# print out more nice stuff, for example 'PASSED' for tests that were fine
brownie test -s
```

The output is the following one:

```
PS C:\Users\<user>\PyScripts\FCC\brownie_SStorage> brownie test
INFORMAZIONI: impossibile trovare file corrispondenti ai
criteri di ricerca indicati.
Brownie v1.18.1 - Python development framework for Ethereum

=============================================================    test    session    starts
=============================================================
platform win32 -- Python 3.9.7, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
rootdir: C:\Users\<user>\PyScripts\FCC\brownie_SStorage
plugins: eth-brownie-1.18.1, anyio-3.5.0, hypothesis-6.27.3, forked-1.4.0, xdist-1.34.0,
web3-5.27.0
collected 2 items

Launching  'ganache-cli.cmd  --port  8545  --gasLimit  12000000  --accounts  10  --hardfork
istanbul --mnemonic brownie'...

tests\test_sstorage.py ..                                   [100%]

=============================================================
2 passed in 16.70s
==============================================================
Terminating local RPC client...
PS C:\Users\<user>\PyScripts\FCC\brownie_SStorage>
```

11.2.3 Networks

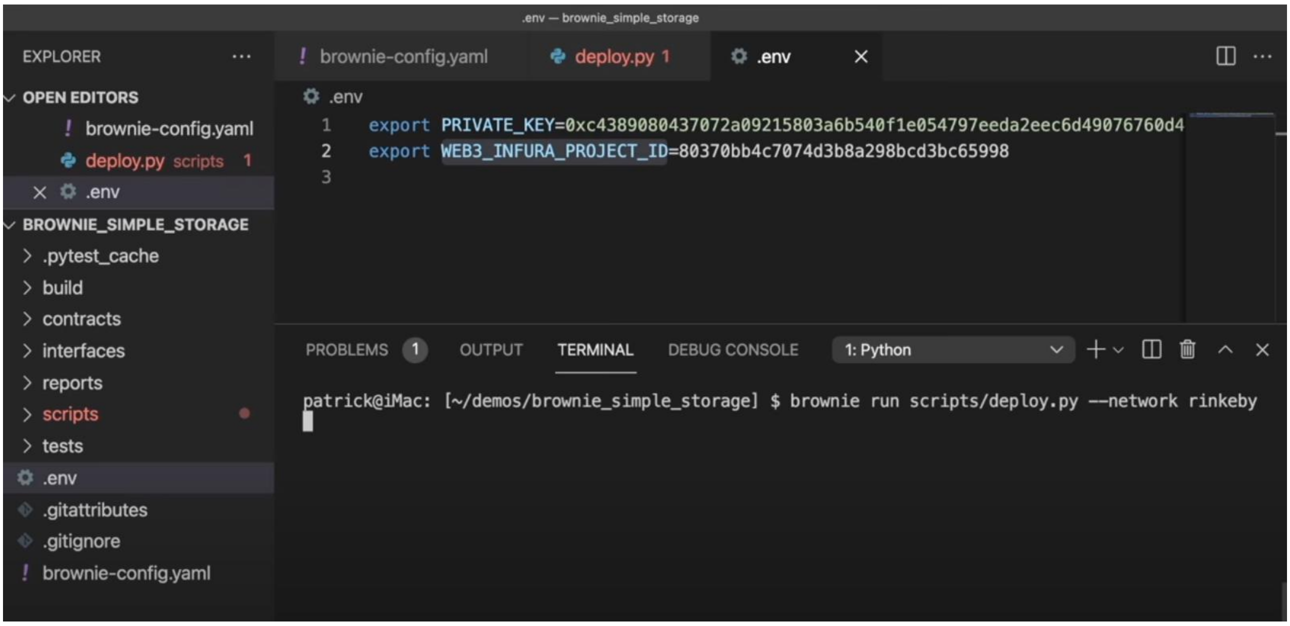
```
PS C:\<path>\brownie_SStorage> brownie networks list
INFORMAZIONI: impossibile trovare file corrispondenti ai
criteri di ricerca indicati.
Brownie v1.18.1 - Python development framework for Ethereum

The following networks are declared:

Ethereum
  ├─Mainnet (Infura): mainnet
  ├─Ropsten (Infura): ropsten
  ├─Rinkeby (Infura): rinkeby
  ├─Goerli (Infura): goerli
  └─Kovan (Infura): kovan

Ethereum Classic
  ├─Mainnet: etc
  └─Kotti: kotti

Arbitrum
  └─Mainnet: arbitrum-main

Avalanche
  ├─Mainnet: avax-main
  └─Testnet: avax-test

Aurora
  └─Mainnet: bsc-main

Fantom Opera
```

```
  ├─Testnet: ftm-test
  └─Mainnet: ftm-main

Harmony
  └─Mainnet (Shard 0): harmony-main

Moonbeam
  └─Mainnet: moonbeam-main

Optimistic Ethereum
  ├─Mainnet: optimism-main
  └─Kovan: optimism-test

Polygon
  ├─Mainnet (Infura): polygon-main
  └─Mumbai Testnet (Infura): polygon-test

XDai
  ├─Mainnet: xdai-main
  └─Testnet: xdai-test

Development
  ├─Ganache-CLI: development
  ├─Geth Dev: geth-dev
  ├─Hardhat: hardhat
  ├─Hardhat (Mainnet Fork): hardhat-fork
  ├─Ganache-CLI (Mainnet Fork): mainnet-fork
  ├─Ganache-CLI (BSC-Mainnet Fork): bsc-main-fork
  ├─Ganache-CLI (FTM-Mainnet Fork): ftm-main-fork
  ├─Ganache-CLI (Polygon-Mainnet Fork): polygon-main-fork
  ├─Ganache-CLI (XDai-Mainnet Fork): xdai-main-fork
  ├─Ganache-CLI (Avax-Mainnet Fork): avax-main-fork
  └─Ganache-CLI (Aurora-Mainnet Fork): aurora-main-fork
PS C:\Users\<user>\PyScripts\FCC\brownie_SStorage>
```

The development networks are torn down right after the deployment.

11.2.4 External networks

To connect to external networks, you need to provide an RPC url, i.e. an url to which can be performed calls to perform what you need to do. In the ".env" file you can do the following:



```
export PRIVATE_KEY=0xc4389080437072a09215803a6b540f1e054797eeda2eec6d49076760d4
export WEB3_INFURA_PROJECT_ID=80370bb4c7074d3b8a298bcd3bc65998
```

```
patrick@iMac: [~/demos/brownie_simple_storage] $ brownie run scripts/deploy.py --network rinkeby
```

Due to the difference network against which we are testing our deployment, we need to adapt our python testing configuration adding the following function:

```
def get_account ():
    if network.show_active() == "development":
        return accounts[0]
    else:
        return accounts.add(config["wallet"]["from_key"])

def test_deploy():
    # arrange
    # act
    # assert
    account = get_account()        <-- change this line too, call the above function
    sstorage = SimpleStorage.deploy({"from": account})
    value = sstorage.retrieve()
    assert (value == 0)
```

In the directory tree under deployment you will see the transaction details, added whenever you use an external blockchain.

11.2.5 Brownie console

```
Brownie console
```

Launches the local ganache, and you can query for variables states, arrays, accounts, smart contracts and so on. You can execute all your scripts line by line, to practice with Ethereum, contracts and so on. It is a very useful feature also for debugging:

```
brownie test --interactive
```

... will open the console when a test fails. See more about console usage on debugging the DAO contract.

11.2.6 Brownie-config.yaml

It's the configuration file for brownie, to read about constraints, dependencies, and other stuff
https://eth-brownie.readthedocs.io/en/stable/config.html

```
dependencies:
    - aragon/aragonOS@4.0.0
    - defi.snakecharmers.eth/compound@1.1.0

dotenv: .env

reports:
    exclude_contracts:
        - SafeMath
        - Owned

reports:
    exclude_paths:
        - contracts/mocks/**/*.*
        - contracts/SafeMath.sol

networks:
    development:
        gas_limit: max
        gas_buffer: 1
        gas_price: 0
        max_fee: null
        priority_fee: null
        reverting_tx_gas_limit: max
        default_contract_owner: true
        cmd_settings:
            port: 8545
            gas_limit: 6721975
            accounts: 10
            chain_id: 1337
            network_id: 1588949648
            evm_version: istanbul
            fork: null
            mnemonic: brownie
            block_time: 0
            default_balance: 100
            time: 2020-05-08T14:54:08+0000
            unlock: null
dependencies:
  - smartcontractkit/chainlink-brownie-contracts@1.1.1
compiler:
    evm_version: null
    solc:
        version: null
        optimizer:
            enabled: true
            runs: 200
        remappings:
          - '@chainlink=smartcontractkit/chainlink-brownie-contracts@1.1.1'
    vyper:
        version: null
```

Dependencies are downloaded from the github repository, and can be found locally under the directory 'build/dependencies'.

11.2.7 Environment variables

Should be put in a ".env" file, to be exported and read on the other scripts. To read things on ethereum scan, you need to register and you can get a token to perform API queries, and use this token as a password, you can use it as usual saving the value in the ".env" file:

```
export PRIVATE_KEY=0xblablabla
export WEB3_INFURA_PROJECT_ID=<infura url>
export ETHERSCAN_TOKEN=<token for Etherscan>
```

11.3 Hardhat

Right now, the hardhat framework is easily the most dominant smart contract development framework. Hardhat is a **javascript and solidity based** development framework that does a beautiful job of quickly getting your applications up to speed. You can check out the hardhat-starter-kit to see an example of what a hardhat project looks like. With Hardhat's testing speed, typescript support, wide adoption, incredible developer experience-focused team, it's no wonder why it's risen so quickly in popularity. At around this time last year, I gave this framework the top spot, and it's still there today. It uses ethersjs on the backend, its own local blockchain for testing, and the team is currently in the midst of building a new cutting edge development platform integrated into Hardhat that I'm BEYOND excited to try for 2022. If you know me, I'm not the biggest fan of javascript due to all its oddities, so often, I prefer to use Hardhat with typescript. Hardhat is easily my second most used framework. I **highly recommend** this framework if you like javascript or you want to use the most popular framework with the most support.

Some of the commands and example have been taken from the following site:
https://dev.to/dabit3/the-complete-guide-to-full-stack-web3-development-4g74

```
npm install [<@scope>/]<pkg>
npm install [<@scope>/]<pkg>@<tag>
npm install [<@scope>/]<pkg>@<version>
npm install [<@scope>/]<pkg>@<version range>
npm install <alias>@npm:<name>
npm install <folder>
npm install <tarball file>
npm install <tarball url>
npm install <git:// url>
npm install <github username>/<github project>

npm install ethers hardhat @nomiclabs/hardhat-waffle
npm install ethereum-waffle chai @nomiclabs/hardhat-ethers
npm install web3modal @walletconnect/web3-provider         <-- this one doesn't work
npm install easymde react-markdown react-simplemde-editor
npm install ipfs-http-client @emotion/css @openzeppelin/contracts
```

The above packages can be installed and be useful to develop the frontend and backend side. To create a new project and a few directories, you can use the following command:

```
npx create-next-app web3-blog
cd web3-blog
# choose 'create e simple project' under the menu
npx hardhat


# performs the scripts contained in the test directory
npx hardhat test

# creates a local Ethereum node with 20 accounts, should be created in a
# separate window to later deploy the contracts
npx hardhat node

#
npx hardhat run scripts/deploy.js --network localhost
```

The difference respect to what we've seen with brownie, is that the language used to deploy and test everything is JAVASCRIPT. You can like or not, if you already know Python it's not that difficult, usually the frontend side for Web3 applications is always written in Javascript. For example this could be the "**test/sample-test.js**" file:

```
const { expect } = require("chai")
const { ethers } = require("hardhat")

describe("Blog", async function () {
  it("Should create a post", async function () {
    const Blog = await ethers.getContractFactory("Blog")
    const blog = await Blog.deploy("My blog")
    await blog.deployed()
    await blog.createPost("My first post", "12345")
    const posts = await blog.fetchPosts()
    expect(posts[0].title).to.equal("My first post")
  })

  it("Should edit a post", async function () {
    const Blog = await ethers.getContractFactory("Blog")
    const blog = await Blog.deploy("My blog")
    await blog.deployed()
    await blog.createPost("My Second post", "12345")
    await blog.updatePost(1, "My updated post", "23456", true)

    posts = await blog.fetchPosts()
    expect(posts[0].title).to.equal("My updated post")
  })

  it("Should add update the name", async function () {
    const Blog = await ethers.getContractFactory("Blog")
    const blog = await Blog.deploy("My blog")
    await blog.deployed()

    expect(await blog.name()).to.equal("My blog")
    await blog.updateName('My new blog')
    expect(await blog.name()).to.equal("My new blog")
  })
})
```

The solidity contract is the following:

```
// contracts/Blog.sol
//SPDX-License-Identifier: Unlicense
pragma solidity ^0.8.0;

import "hardhat/console.sol";
import "@openzeppelin/contracts/utils/Counters.sol";

contract Blog {
  string public name;
  address public owner;

  using Counters for Counters.Counter;
  Counters.Counter private _postIds;

  struct Post {
   uint id;
   string title;
   string content;
   bool published;
  }
  /* mappings can be seen as hash tables */
  /* here we create lookups for posts by id and posts by ipfs hash */
  mapping(uint => Post) private idToPost;
  mapping(string => Post) private hashToPost;

  /* events facilitate communication between smart contractsand their user interfaces  */
  /* i.e. we can create listeners for events in the client and also use them in The Graph  */
  event PostCreated(uint id, string title, string hash);
  event PostUpdated(uint id, string title, string hash, bool published);

  /* when the blog is deployed, give it a name */
  /* also set the creator as the owner of the contract */
  constructor(string memory _name) {
    console.log("Deploying Blog with name:", _name);
    name = _name;
    owner = msg.sender;
  }

  /* updates the blog name */
  function updateName(string memory _name) public {
    name = _name;
  }

  /* transfers ownership of the contract to another address */
  function transferOwnership(address newOwner) public onlyOwner {
    owner = newOwner;
  }
  /* fetches an individual post by the content hash */
  function fetchPost(string memory hash) public view returns(Post memory){
    return hashToPost[hash];
  }

  /* creates a new post */
  function createPost(string memory title, string memory hash) public onlyOwner {
      _postIds.increment();
      uint postId = _postIds.current();
      Post storage post = idToPost[postId];
      post.id = postId;
      post.title = title;
      post.published = true;
      post.content = hash;
      hashToPost[hash] = post;
      emit PostCreated(postId, title, hash);
  }

  /* updates an existing post */
  function updatePost(uint postId, string memory title, string memory hash, bool published) public
onlyOwner {
      Post storage post =  idToPost[postId];
      post.title = title;
      post.published = published;
      post.content = hash;
      idToPost[postId] = post;
      hashToPost[hash] = post;
      emit PostUpdated(post.id, title, hash, published);
  }

  /* fetches all posts */
  function fetchPosts() public view returns (Post[] memory) {
      uint itemCount = _postIds.current();
      uint currentIndex = 0;

      Post[] memory posts = new Post[](itemCount);
      for (uint i = 0; i < itemCount; i++) {
          uint currentId = i + 1;
          Post storage currentItem = idToPost[currentId];
          posts[currentIndex] = currentItem;
          currentIndex += 1;
      }
      return posts;
  }

  /* this modifier means only the contract owner can */
  /* invoke the function */
  modifier onlyOwner() {
    require(msg.sender == owner);
  _;
 }
}
```

This is not a real example, it's a Web3 hypotetical blog. On a real blockchain it could be too expensive, since there's too much data to be put on the global blockchain.

11.4 Truffle

This is another suite based on other software packages, like for example NodeJs and Ganache. The 'DappUniversity' guy on the web uses this tool, developed and maintained by ConsenSys company. Scripts to deploy contracts are written in Javascript, so you need to be familiar with this language.

```
npm install -g truffle@version
# creates directories inside a project
truffle init
truffle migrate -reset
github clone https://github.com/dappuniversity/blockchain_game
```

In the main directory you have a 'Truffle-Config.js' file like the following one:

```
require('babel-register');
require('babel-polyfill');

module.exports = {
  networks: {
    development: {
      host: "127.0.0.1",          <-- local Ganache instance to be manually launched
      port: 8545,
      network_id: "*" // Match any network id
    },
  },
  contracts_directory: './src/contracts/',        <-- change default dir
  contracts_build_directory: './src/abis/',       <-- change default dir
  compilers: {
    solc: {
      optimizer: {
        enabled: true,
        runs: 200
      }
    }
  }
}
```

```
PS C:\Users\<user>\PyScripts> truffle.cmd migrate --reset
Could not find suitable configuration file.
Truffle v5.5.3 (core: 5.5.3)
Node v16.13.1

PS C:\Users\<user>\PyScripts> cd .\fcc\blockchain_game\
PS C:\Users\<user>\PyScripts\fcc\blockchain_game> truffle.cmd migrate --reset

Compiling your contracts...
===========================
> Compiling .\src\contracts\ERC721Full.sol
> Compiling .\src\contracts\MemoryToken.sol
> Compiling .\src\contracts\Migrations.sol
> Artifacts written to C:\Users\<user>\PyScripts\fcc\blockchain_game\src\abis
> Compiled successfully using:
   - solc: 0.5.16+commit.9c3226ce.Emscripten.clang
> Something went wrong while attempting to connect to the network at http://127.0.0.1:8545.
Check your network configuration.

Could not connect to your Ethereum client with the following parameters:
    - host       > 127.0.0.1
    - port       > 8545
    - network_id > *
Please check that your Ethereum client:
    - is running
    - is accepting RPC connections (i.e., "--rpc" or "--http" option is used in geth)
    - is accessible over the network
    - is properly configured in your Truffle configuration file (truffle-config.js)

Truffle v5.5.3 (core: 5.5.3)
Node v16.13.1
PS C:\Users\<user>\PyScripts\fcc\blockchain_game>
```

Ganache has to be manually launched, differently from the other tools we saw. Maybe it's also something configurable.

Deploy **scripts** are contained into **'migrations'** directory:

```
# file "1_initial_migration.js"
const Migrations = artifacts.require("Migrations");

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

# file "2_deploy_contracts.js"
const MemoryToken = artifacts.require("MemoryToken");

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

If you launch the local Ganache, this is the output:

```
PS C:\Users\<user>\PyScripts\fcc\blockchain_game> truffle.cmd migrate --reset

Compiling your contracts...
===========================
> Compiling .\src\contracts\ERC721Full.sol
> Compiling .\src\contracts\MemoryToken.sol
> Compiling .\src\contracts\Migrations.sol
> Artifacts written to C:\Users\<user>\PyScripts\fcc\blockchain_game\src\abis
> Compiled successfully using:
   - solc: 0.5.16+commit.9c3226ce.Emscripten.clang


Starting migrations...
======================
> Network name:    'development'
> Network id:      1337
> Block gas limit: 6721975 (0x6691b7)


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

   Replacing 'Migrations'
   ----------------------
   >                                  transaction                                   hash:
0x6eacc4fe83b9a96b833e8822df94099bb7431ffea34f9c5e0183d7402222468a
   > Blocks: 0            Seconds: 0
   > contract address:    0x55C4dfF06fd6CA67847Dfb50dD010A00b2254bD7
   > block number:        1
   > block timestamp:     1646837727
   > account:             0xeaAe2D4Af70802c81015bf0651C96fC277a29460
   > balance:             99.99549526
   > gas used:            225237 (0x36fd5)
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.00450474 ETH

   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:          0.00450474 ETH


2_deploy_contracts.js
=====================

   Replacing 'MemoryToken'
   -----------------------
   >                                   transaction                                   hash:
0xfba3eae8bbe528303a018aea0e9b9abe4e3fd74350dda5b9d19953f69360c8f5
   > Blocks: 0            Seconds: 0
   > contract address:    0x82190D355c69DCED1921fDf117dC6C4Fe71cF394
   > block number:        3
   > block timestamp:     1646837731
   > account:             0xeaAe2D4Af70802c81015bf0651C96fC277a29460
   > balance:             99.9495942
   > gas used:            2252690 (0x225f92)
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.0450538 ETH

   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:           0.0450538 ETH

Summary
=======
> Total deployments:   2
> Final cost:          0.04955854 ETH
```

Under the directory ‘test’ you can find the file ‘MemoryTokenTest.js’:

```
const MemoryToken = artifacts.require('./MemoryToken.sol')

require('chai')
  .use(require('chai-as-promised'))
  .should()

contract('Memory Token', (accounts) => {
  let token

  before(async () => {
    token = await MemoryToken.deployed()
  })

  describe('deployment', async () => {
    it('deploys successfully', async () => {
      const address = token.address
      assert.notEqual(address, 0x0)
      assert.notEqual(address, '')
      assert.notEqual(address, null)
      assert.notEqual(address, undefined)
    })

    it('has a name', async () => {
      const name = await token.name()
      assert.equal(name, 'Memory Token')
    })

    it('has a symbol', async () => {
      const symbol = await token.symbol()
      assert.equal(symbol, 'MEMORY')
    })
  })

  describe('token distribution', async () => {
    let result

    it('mints tokens', async () => {
      await token.mint(accounts[0], 'https://www.token-uri.com/nft')

      // It should increase the total supply
      result = await token.totalSupply()
      assert.equal(result.toString(), '1', 'total supply is correct')

      // It increments owner balance
      result = await token.balanceOf(accounts[0])
      assert.equal(result.toString(), '1', 'balanceOf is correct')

      // Token should belong to owner
      result = await token.ownerOf('1')
      assert.equal(result.toString(), accounts[0].toString(), 'ownerOf is correct')
      result = await token.tokenOfOwnerByIndex(accounts[0], 0)

      // Owner can see all tokens
      let balanceOf = await token.balanceOf(accounts[0])
      let tokenIds = []
      for (let i = 0; i < balanceOf; i++) {
        let id = await token.tokenOfOwnerByIndex(accounts[0], i)
        tokenIds.push(id.toString())
      }
      let expected = ['1']
      assert.equal(tokenIds.toString(), expected.toString(), 'tokenIds are correct')

      // Token URI Correct
      let tokenURI = await token.tokenURI('1')
      assert.equal(tokenURI, 'https://www.token-uri.com/nft')
    })
  })
})
```

The output of the tests is the following:

```
PS C:\Users\<user>\PyScripts\fcc\blockchain_game> truffle.cmd test
Using network 'development'.

Compiling your contracts...
===========================
> Compiling .\src\contracts\ERC721Full.sol
> Compiling .\src\contracts\MemoryToken.sol
> Compiling .\src\contracts\Migrations.sol
> Artifacts written to C:\Users\<user>\AppData\Local\Temp\test--4376-HjDuELBffLHD
> Compiled successfully using:
   - solc: 0.5.16+commit.9c3226ce.Emscripten.clang

  Contract: Memory Token
    deployment
      √ deploys successfully
      √ has a name (527ms)
      √ has a symbol (204ms)
    token distribution
      √ mints tokens (2507ms)

  4 passing (5s)
```

12 Calls and transactions

12.1 Call

A call is a local invocation of a contract function that does not broadcast or publish anything on the blockchain. It is a read-only operation and will not consume any Ether. It simulates what would happen in a transaction, but discards all the state changes when it is done. It is synchronous and the return value of the contract function is returned immediately. Its web3.js API is web3.eth.call and is what's used for **Solidity** view**,** pure**,** constant **functions**. Its underlying JSON-RPC is eth_call.

12.2 Transaction

A transaction is broadcasted to the network, processed by miners, and if valid, is published on the blockchain. It is a write-operation that will affect other accounts, update the state of the blockchain, and consume Ether (unless a miner accepts it with a gas price of zero). It is asynchronous, because it is possible that no miners will include the transaction in a block (for example, the gas price for the transaction may be too low). Since it is asynchronous, the immediate **return value of a transaction is always the transaction's hash.** To get the "return value" of a transaction to a function, Events need to be used (unless it's Case4 discussed below). For ethers.js an example: listening to contract events using ethers.js?
Its web3.js API is web3.eth.sendTransaction and is used if a Solidity function is not marked constant. Its underlying JSON-RPC is eth_sendTransaction. sendTransaction will be used when a verb is needed, since it is clearer than simply transaction.

12.3 Recommendation to Call first, then sendTransaction

Since a sendTransaction costs Ether, it is a good practice to "test the waters" by issuing a *call* first, before the sendTransaction. This is a free way to debug and estimate if there will be any problems with the sendTransaction, for example if an Out of Gas exception will be encountered.
This "dry-run" usually works well, but in some cases be aware that *call* is an estimate, for example a contract function that returns the previous blockhash, will return different results based on when the *call* was performed, and when the transaction is actually mined.
Finally, note that even though a *call* does not consume any Ether, sometimes it may be necessary to specify the actual gas amount for the *call*: the default gas for *call* in clients such as Geth, may still be insufficient and can still lead to Out of Gas.

13 Brownie mixes and Chainlink mix

On github you can search for ‘brownie mix’, and you can find many useful repos. To clone one of them you can use the following command, everything will be copied starting from the present directory. It doesn’t make any sense to rewrite always everything, it’s important to understand Solidity and what it does, but public contracts are usually audited by more eyes and people, thus leading to more secure and safe code, that can be re-used and extended through inheritance and polymorphism. Beware that it’s a best practice to have ‘smart contracts’ published on the internet, so that everyone can have a look to the code, but it’s not mandatory, and on the blockchain you can’t see the source code, is only **pushed the bytecode**, which is **not human readable**.

```
brownie bake chainlink-mix                <-- like git clone

cd chainlink
brownie.exe test
```

14 Github

It’s far away the most used versioning and control software, even though it has been acquired by Microsoft. It can be used online, directly on:

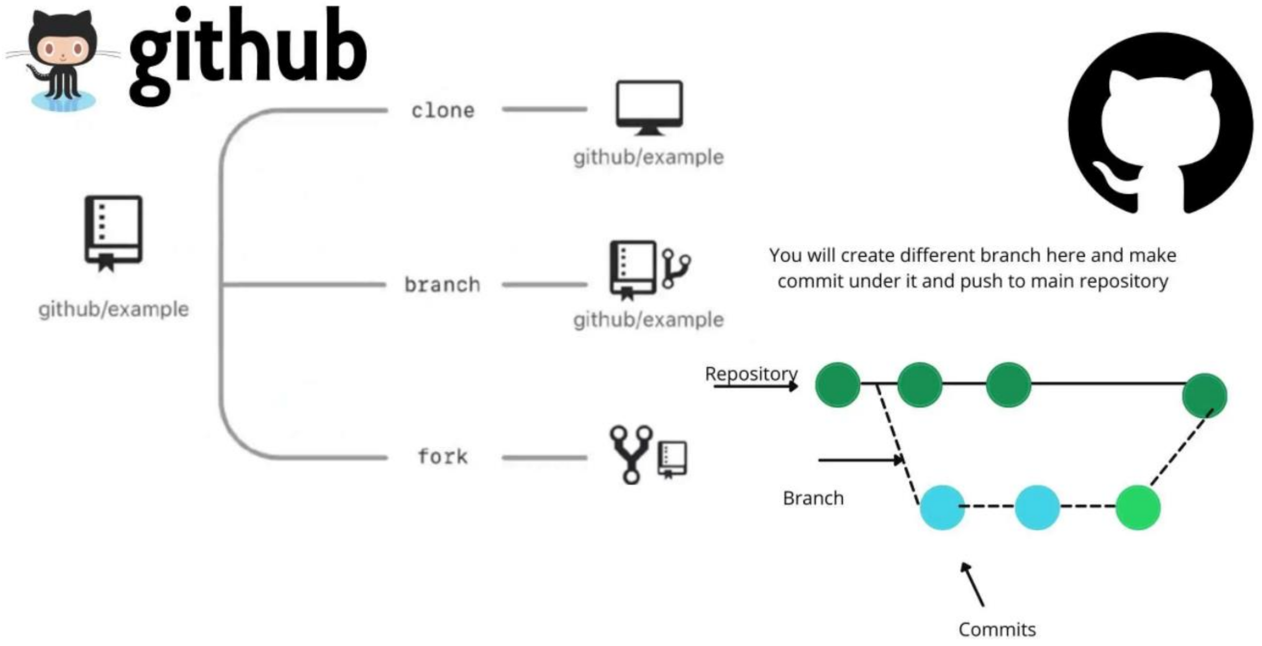
www.github.com

... or with a software IDE, or through the command line, which everyone (used to CLI) will probably find the fastest and easiest way to keep everything in sync.

Remember to create a '.gitignore' file to list all the folders that MUST NOT be uploaded to github, because they contain private data and passwords, or because they are public libraries or built stuff that should not be exported. It's a simple list like the following one, wildcard characters can be used:

```
.gitignore
.env
*.log
/<directory-to-be-ignored>
/build
/artifacts
```

This software has been thought to manage different versions of the files, 'branches' commits and so on. Imagine a central repository with many people working on it, from different sites in the world. All these people can **clone** the project locally in their personal computer and open a new 'branch'. They modify some files, test the new version, modify the files again. When the job is done, they upload the new version on the central repository, potentially keeping the branch name. Then someone responsible for this critical activity, will MERGE the branch on the main branch, solving all potential conflicts, watching the differences between the files, choosing the right code. This is because more than a single developer could have changed the same file, thus the merge could be the fusion of the work of 2 persons on the same file. This will help keeping track of every modification, potentially rolling back on a previous release if necessary. Professional companies that develop software MUST necessarily use such tools to coordinate hundreds of people working on the same project.

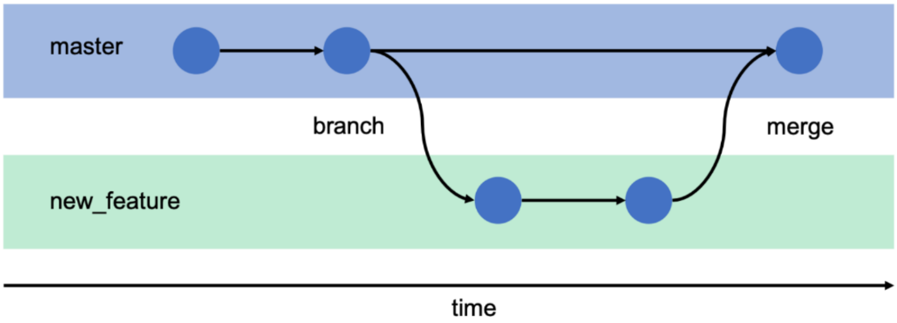

github
clone
github/example
github/example
branch
github/example
fork
You will create different branch here and make commit under it and push to main repository
Repository
Branch
Commits

In our case, probably projects will be smaller and maybe managed just by us.

```
github clone https://github.com/<path to github project>
```

This command is used for downloading the latest version of a remote project and copying it to the selected location on the local machine. It looks like this:

```
git clone <repository url>
git clone <repository url> -b <branch name>        <-already open a new branch
```



```
git init <repository name>
```

This is the command you need to use if you want to start a new empty repository or to reinitialize an existing one in the project root. It will create a .git directory with its subdirectories. To add local files or directories to local repository, use the following commands:

```
git add <file name>
git add .
```

Store changes so that they can be pushed on remote repository (adding a comment is):

```
git commit –m "comment to commit"         <- store changes locally
```

Show the present status of git files of the local repository:

```
git status                 <- shows the current status, added/removed files
```

This command is necessary to create a 'link' between the local repository and the remote repository. Beware that the local and remote repository names DO NOT have to be necessarily the same, even though it's probably better to avoid confusion. Beware that the remote repository needs to be already there, 'origin' is the name that refers from now on to the remote repository:

```
git remote add origin http://github.com/ricky-andre/Bitcoin.git
```

The following commands are used to retrieve the remote repository presently active, and where a 'push' command would try to put the new local files that have been committed:

```
git config --get remote.origin.url
git remote show origin
git remote -v
```

If you have cloned a repository, you have changed it, and you want to upload the files into your personal repository, you will need to remove the origin and re-add the target one.

```
git remote remove origin
git remote add origin http://github.com/<your repository name>.git
git remote rename <old_name> <new_name>

git branch -M main         <- crea un branch di nome 'main' invece che 'master'
```

The following command should do the job of uploading all files to Github repository (with one single command):

```
git push -u origin master
git push -u origin <branch name>
git push         <- push the current branch
```

In case of errors, use ‘git status’ and check for uncommited changes, commit them in case of errors. Beware that in case the remote repository already exists and you would overwrite some of the remote files, you get an error unless you manually remove the remote files or use the `‘--force’` command option. Other useful push commands are the following:

```
git push --set-upstream <remote branch> <branch name>
```

To perform a checkout and create new branches:

```
git checkout <branch name>
```

In case things have been changed in the main repository and also on your local repository, you could try to understand the changes in the following way:

```
git diff
git diff --staged
gif diff <branch1> <branch2>
```

Using **git pull** will fetch all the changes from the remote repository and merge any remote changes in the current local branch. This command will **NOT** cancel nor overwrite local files that have the same name respect to remote files, unless you explicitly configure the `‘--force’` option.

```
git pull REMOTE-NAME BRANCH-NAME
```