Contents

[**Security Tools Lab 2** 1](#_Toc85564400)

[Project 2 – DAPP Distributed Storage 1](#_Toc85564401)

[**Introduction** 3](#_Toc85564402)

[Ethereum 3](#_Toc85564403)

[IPFS 3](#_Toc85564404)

[IPFS and Ethereum 3](#_Toc85564405)

[**Methodology** 4](#_Toc85564406)

[Approach 4](#_Toc85564407)

[**Collecting Test ether for Ropsten** 4](#_Toc85564408)

[**Deployment of Smart contract** 5](#_Toc85564409)

[**React Web Application** 6](#_Toc85564410)

[Setup Environment 6](#_Toc85564411)

[getWeb3.js 8](#_Toc85564412)

[ipfs.js 8](#_Toc85564413)

[App.css 9](#_Toc85564414)

[App.js 9](#_Toc85564415)

[**Demo** 12](#_Toc85564416)

[Production Deployment 13](#_Toc85564417)

## **Introduction**

### Ethereum

It is extremely expensive to store huge amounts of data on the Ethereum blockchain. As per Ethereum’s Yellow Paper, it costs 20,000 gas for a 256-bit word. This means a kilobyte worth of storage will cost 640,000 gas, which corresponds to 0.0576 ETH, or US$201.60[[1]](#footnote-1).  
These costs are expected to rise exponentially, given both the increasing adoption of Ethereum as well as the network congestion which will rise further as a result of said adoption. Already, present gas costs are excessively prohibitive, giving rise to competing layer 1 blockchains such as Polkadot or Cardano.

It is very clear that there is no sense in using Ethereum to store data. If we can only afford to store a couple of kilobytes of data on the blockchain, then we have to look to other solutions to be used in tandem with Ethereum as a distributed storage solution. Here, we will explore how we can achieve building a Dapp for distributed storage by using a decentralized network called the InterPlanetary File System (“IPFS”) together with Ethereum.

### IPFS

IPFS is a decentralized storage and file referencing system for Web3. IPFS is a versioned file system at its core, which can store files and track versions over time, much like git. IPFS also defines how a file moves over the network, making it a distributed file system.

Presently, when a user wants to download a file off the Internet, the user has to query the exact address of the file – which is the IP address of the server. This is known as located-based addressing. If the server is offline, the user will be unable to access the file, even if there exist multiple copies of the very same file at other locations.

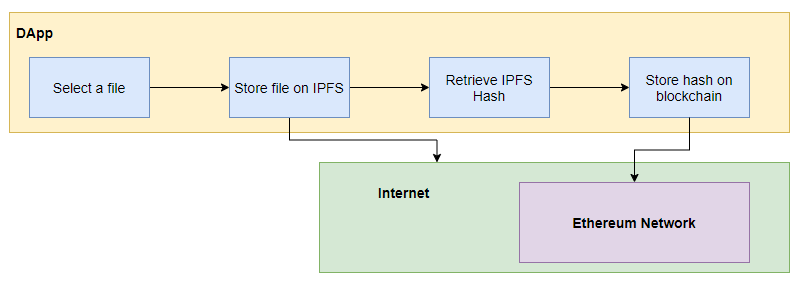
In contrast, IPFS uses content-based addressing. Every file that has been uploaded is associated with its unique hash, and therefore there are no duplicates maintained. This keeps the network every efficient. Additionally, the user can simply query the file to obtain using its hash and the closest peers will provide it.

### IPFS and Ethereum

Blockchains like Ethereum provide us with immutable timestamped data, which makes tampering with that data almost impossible. But to store a file on Ethereum can be very expensive given the amount of processing the Ethereum blockchain has to do, whereas the content-based addresses that IPFS returns are 46 bytes in size which is a tiny amount considering the size of files. So, it is cheaper to publish a file on IPFS and store their content-based address on the Ethereum blockchain.

## **Methodology**

We will select a file, publish it on IPFS, retrieve the hash of this file and store it on the Ethereum network. The hash of a file on IPFS can be thought of as the URL to retrieve it.



### Approach

1. Collect test ether from faucet
2. Deploy smart contract on Ropsten testnet
3. Create a React frontend for UIUX
4. Deploy DApp to Netlify (free web application hosting)

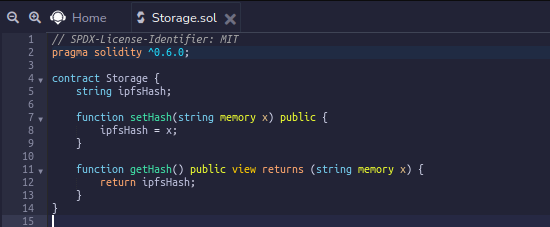
## **Collecting Test ether for Ropsten**

We will need test ether to successfully deploy our smart contract onto the public testnet. Here we shall deploy it to the Ropsten network.

Therefore, we will need to obtain test ether on the Ropsten network. After successfully creating a Metamask wallet, we will navigate to a Ropsten Ethereum Faucet, <https://faucet.ropsten.be/> to receive test ether.

## **Deployment of Smart contract**

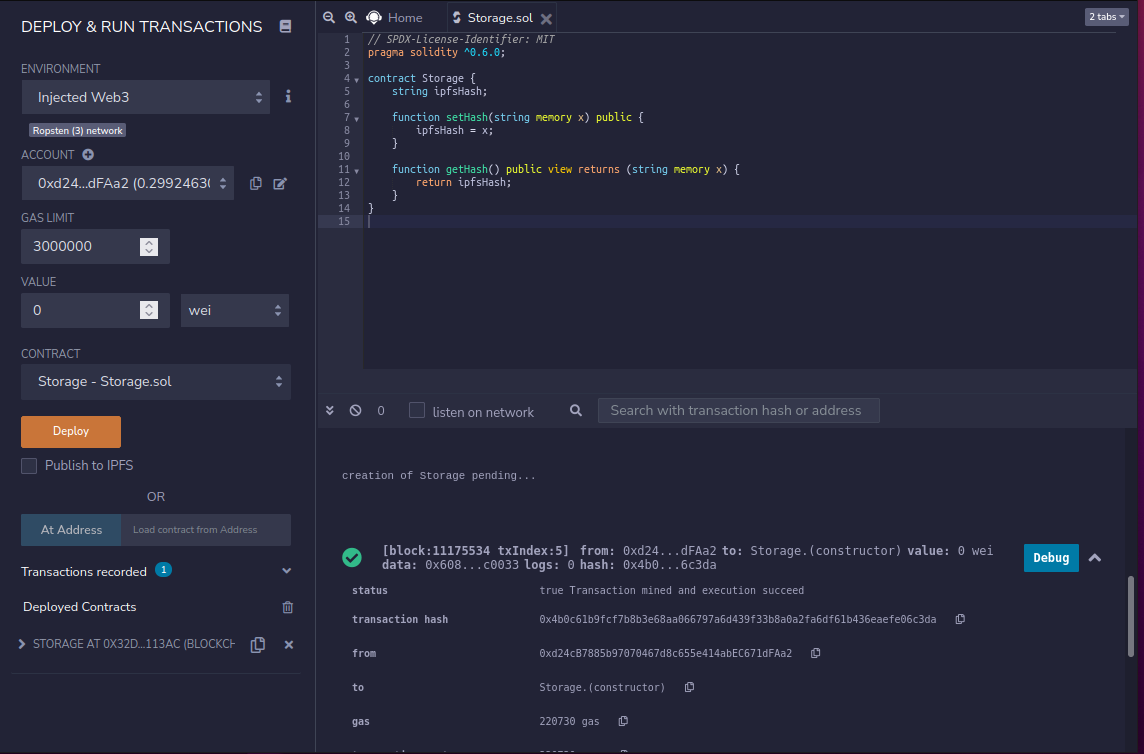
To deploy our smart contract, Storage.sol, on the blockchain we will use Remix. The contract is saved as Storage.sol for reference.



The pragma statement specifies which solidity compiler version to use.

The function setHash receives a string x as argument and stores it in the variable ipfsHash, thereby storing it on the blockchain. The function getHash returns the string value stored in the variable ipfsHash.

We will compile the contract and copy its ABI (Application Binary Interface). **This is the information with which we call functions in a smart contract from a frontend.** The ABI is also saved as ABI.txt file for reference.



When deploying we ensure that in the dropdown menu for the field ‘ENVIRONMENT’ on the left column, “Injected Web3” is selected. This will pick up which network our Metamask wallet has been set to – in this case, the Ropsten network.

After the transaction has been processed to be included in a block and recorded on blockchain, Remix will display the address where the contract is deployed. In our deployment the smart contract address is at: 0x32D64855f891BcdeA2CaD990139904b0401113aC.

The transaction for this deployment can we viewed on etherscan.io at: <https://ropsten.etherscan.io/tx/0x4b0c61b9fcf7b8b3e68aa066797a6d439f33b8a0a2fa6df61b436eaefe06c3da>

## **React Web Application**

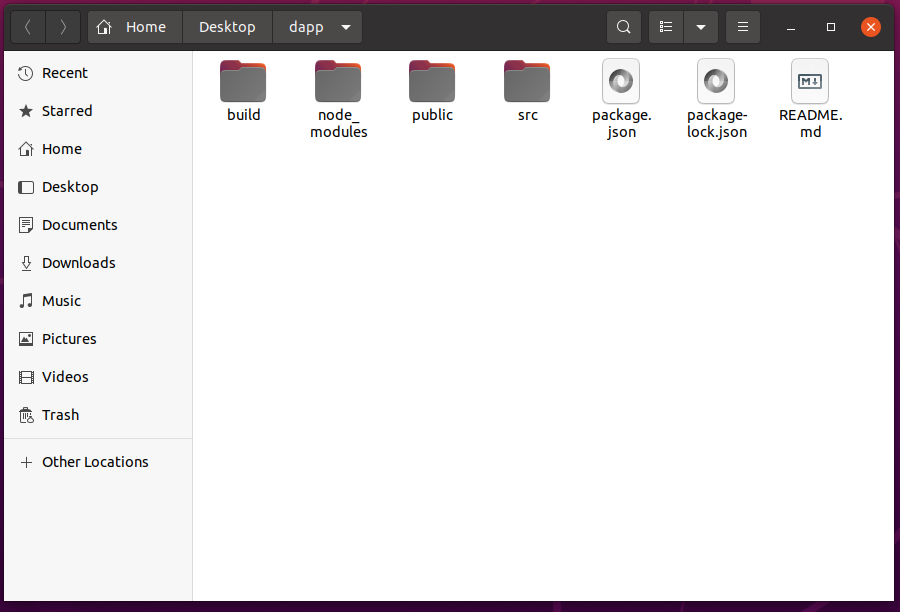
### Setup Environment

Next, we will construct a React web application via a boilerplate by executing a nodeJS module named create-react-app. Then we import the other necessary nodeJS dependencies.

We run the command as below to generate the boilerplate code for the react web application:

$ npx create-react-app dapp

This will create a folder named dapp that will contain all the JavaScript packages and code to make a basic React application. The folder src as contained within the dapp folder will encompass our application.



Move inside the working folder to install dependencies:

$ cd dapp

Add CSS framework and its React helper

$ npm install react-bootstrap bootstrap

Add the waiting spinner during a transaction

$ npm install react-spinners

Add access to local file system:

$ npm install fs-extra

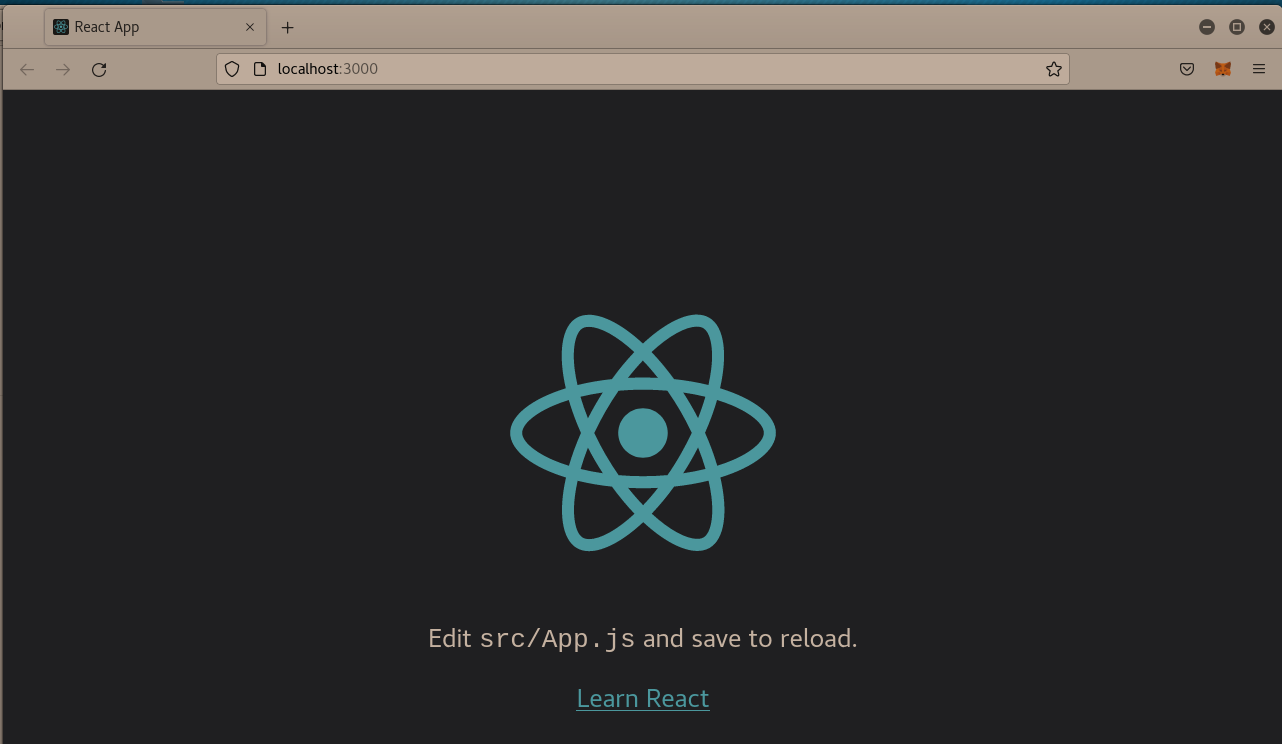
Add IPFS access and blockchain access:

$ npm install ipfs-http-client web3

Once these dependencies have been installed, we can run the React app with:

$ npm start

Create-React-App should automatically render on <http://localhost:3000/> as we see below:



There a number of files we need to create and modify. Inside our “dapp/src” directory, we create the following files.

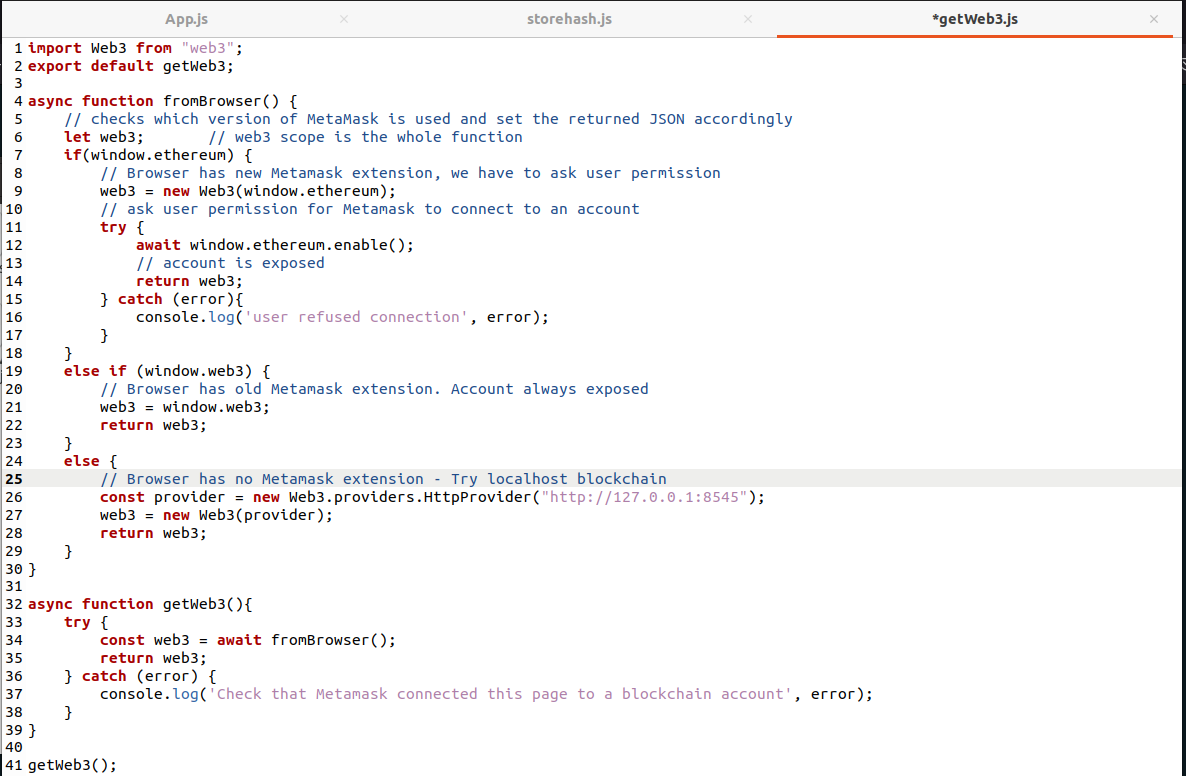
1. getWeb3.js
2. ipfs.js
3. storehash.js

### getWeb3.js

We need to create this file within the src directory. It contains a component that returns a JavaScript object. The key/value pairs of this object describe the API calls to a blockchain node that do transactions.

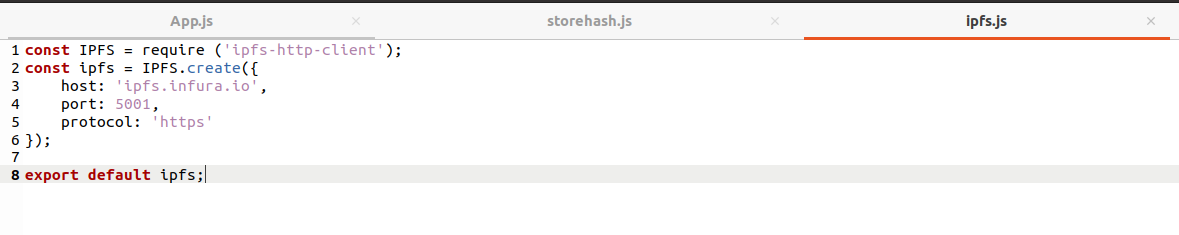
The logic of these functions covers the cases where the DApp is called from the new version of MetaMask, from an old version of MetaMask and where the DApp runs without MetaMask on a local test blockchain network.

In summary, we check if window.ethereum exists, then create a window.web3 object, using the window.ethereum object as the input provider. The window.ethereum.enable() line allows for the UI pop-up request to connect your Metamask to the DApp. The full code is commented out in detail as below:



### ipfs.js

We will run the ipfs.infura.io node to connect to IPFS instead of running an IPFS daemon on our own computer. We export a JavaScript object with the API description to do IPFS transactions.



**storehash.js**

This file contains a function that creates the JavaScript instance connecting with the smart contract we deployed earlier. We need to input our contract address and its corresponding ABI information.



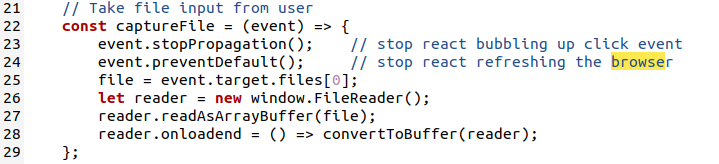
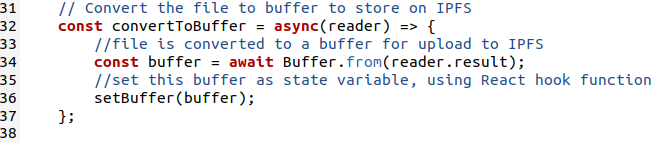
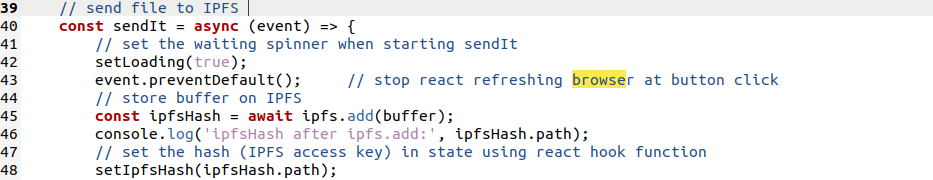
Finally, we modify the App.js and App.css that were created by create-react-app. The bulk of our code will be in App.js.

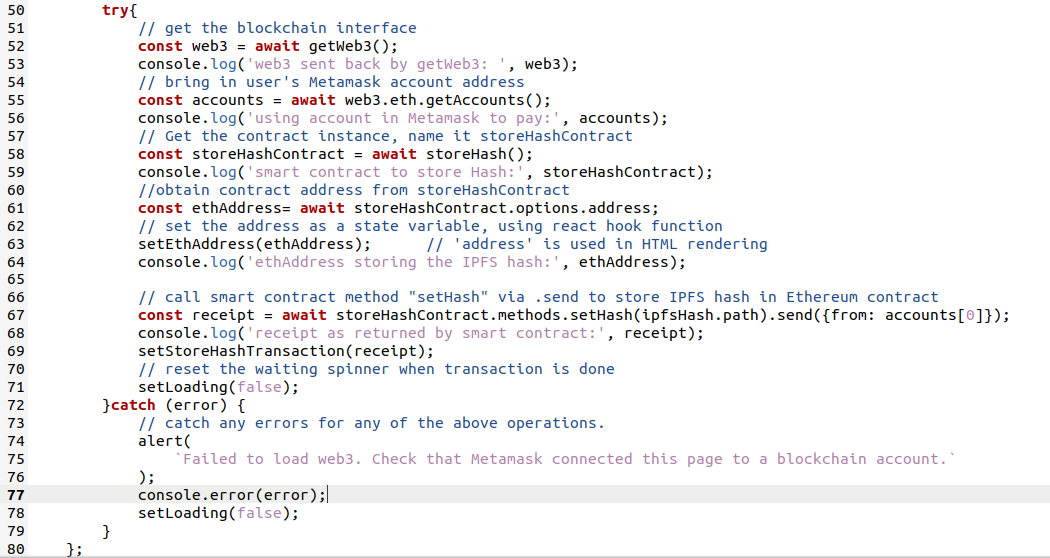
### App.css

Editing this allows us to modify the UIUX aspects of our DApp. We can change font sizes, physical dimensions and the like.

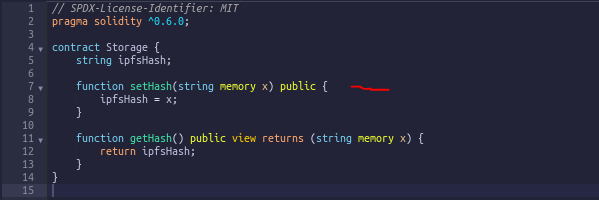
### App.js

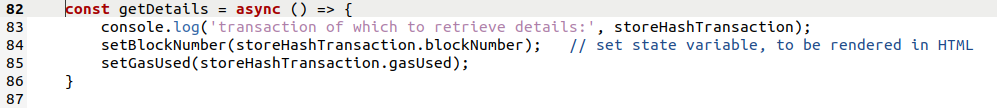
This class is the main application that uses React. Its template was created by create-react-app. We will change this heavily. The full code of the file is rather long, so we shall opt to summarise its order of operations as follows.

1. Declare the state variables
2. Capture the file input from user  
   
   1. This allows us to use a file explorer pop-up to select our desired file
3. Convert the selected file to a buffer for IPFS storage
   1. 
   2. IPFS expects you to read the file into a Buffer (an array with data).
   3. A Buffer is just a raw collection of bytes and as such doesn't make any assumptions about encodings or the type of data it contains.
4. Send the buffered file to IPFS and IPFS returns the unique hash of the file  
   
5. Obtain the User’s Metamask Ethereum wallet address and,
6. Store IPFS hash in Ethereum contract

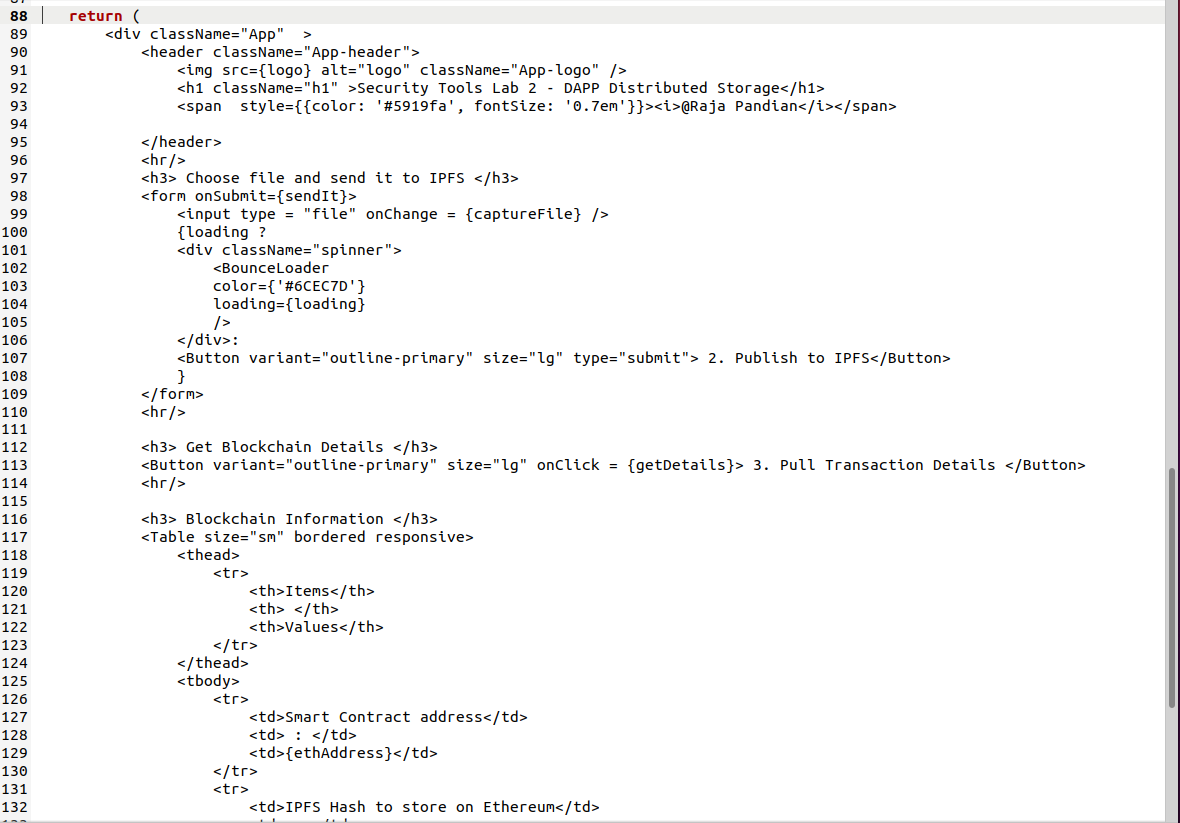


We call on the very same “setHash” function above as we set in our Solidity smart contract:



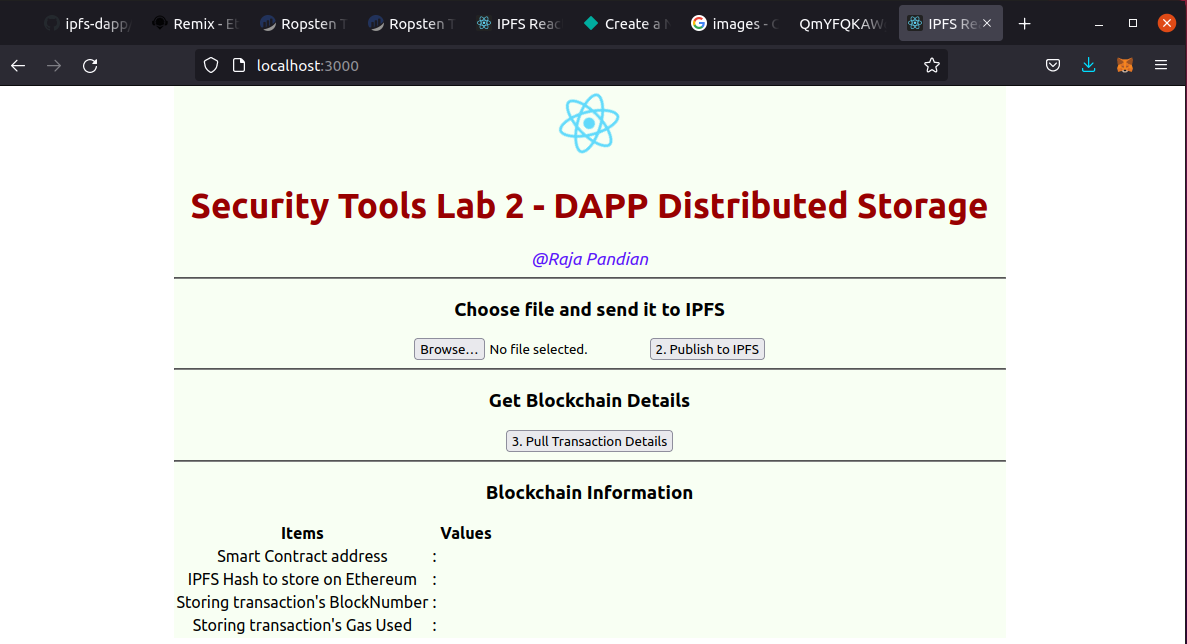
1. On success, we are returned a hash number for the transaction on the blockchain
2. The transaction hash number can be used to obtain a transaction receipt containing details like the block number and gas used.  
   

The code from line 88 onwards allows us to change what we render on the UI. Here, we create a simple form and a submit button.

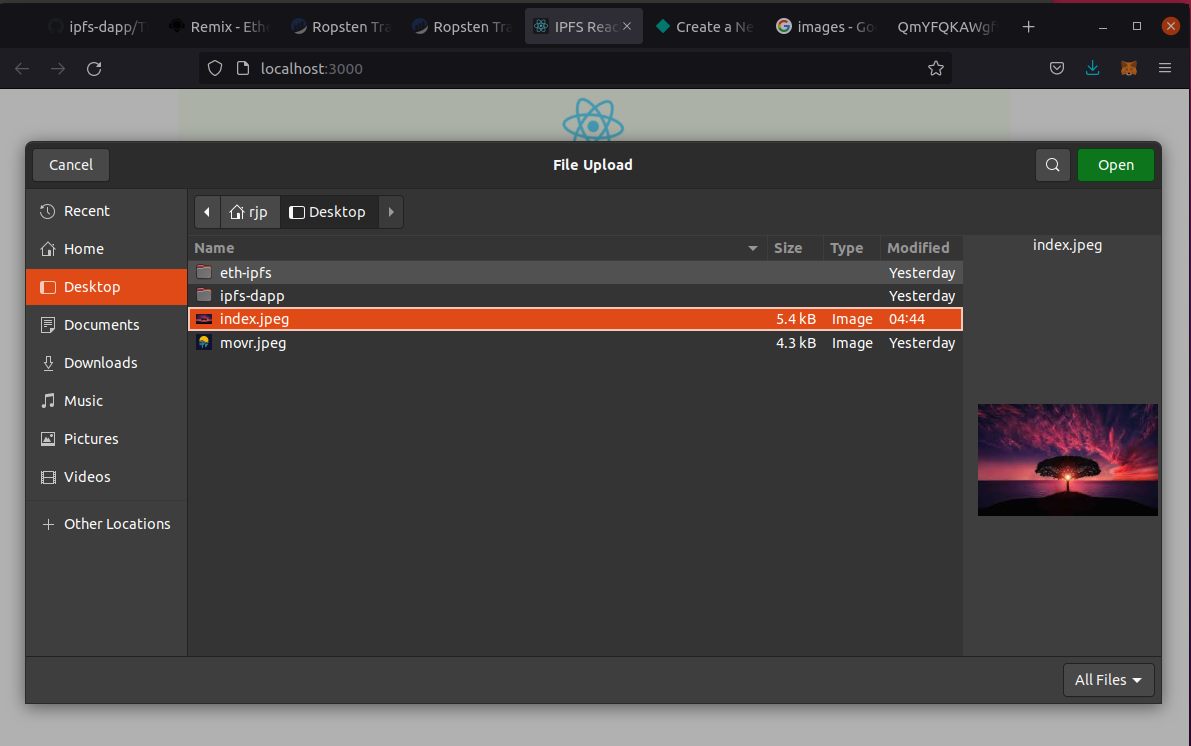


## **Demo**

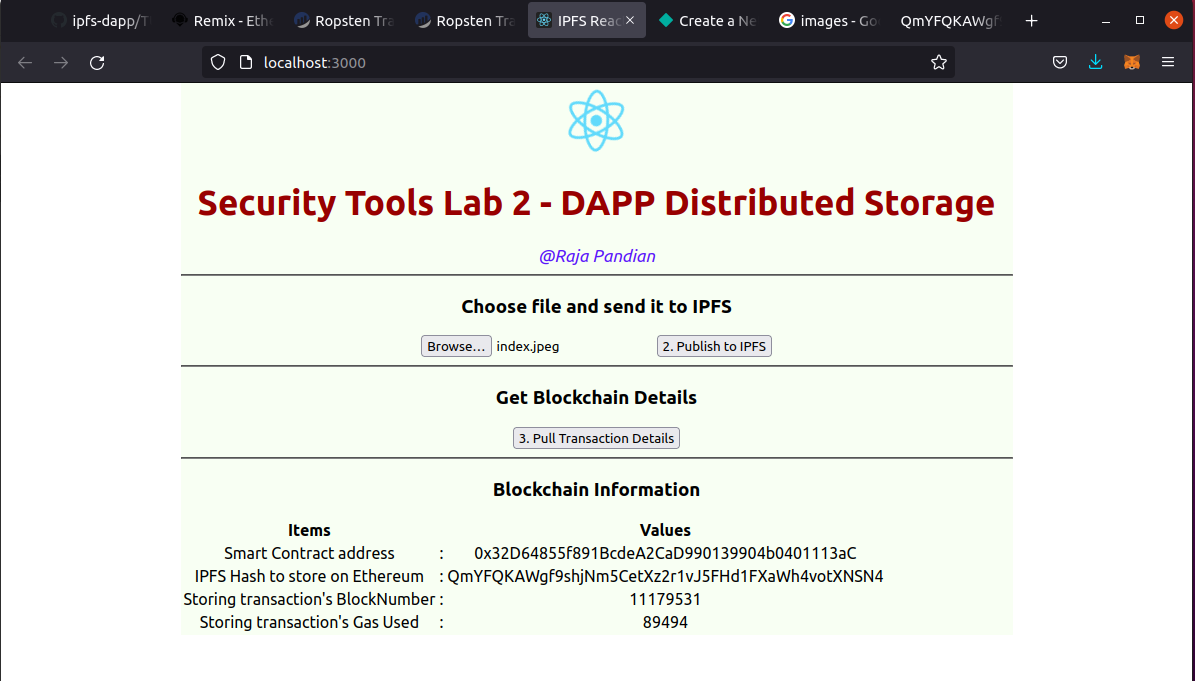
We type npm start within the dapp folder to launch the react app and we see the result in our browser as below:



We shall select an image file, index.jpeg to upload.

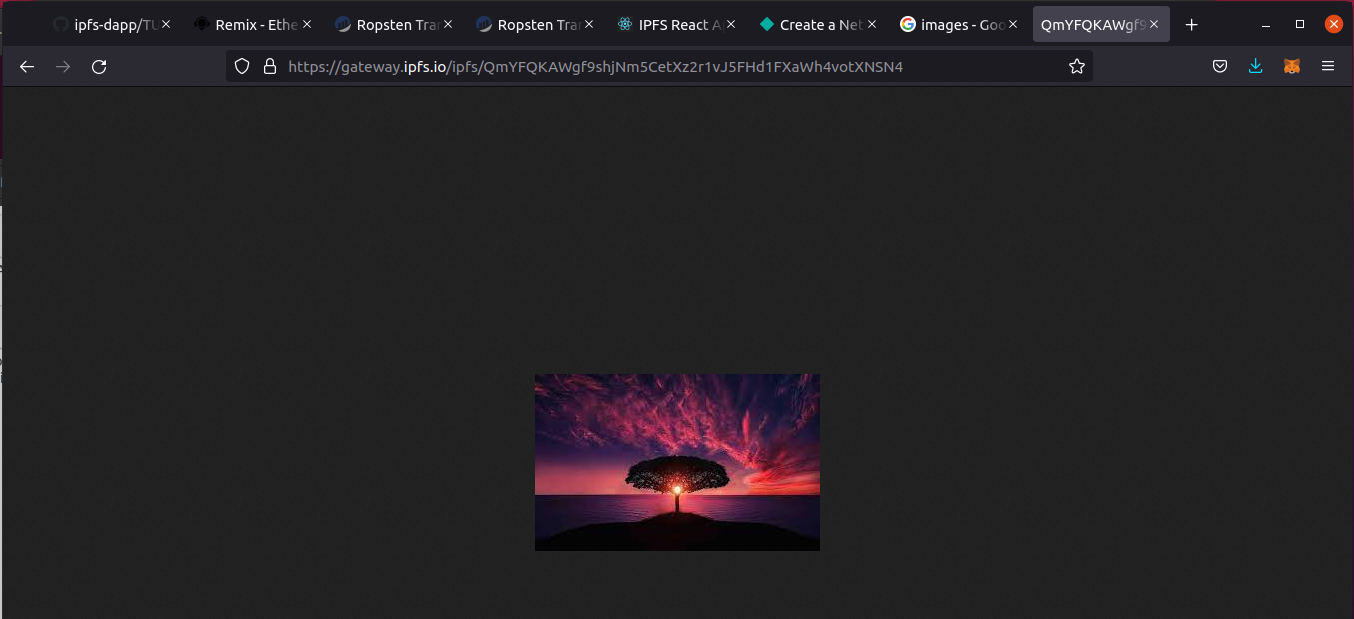


After selecting the file and clicking the second button, “Publish to IPFS”, we see the results as below.



Our smart contract address corresponds to that of the contract we deployed earlier on the Ropsten network. We obtain blockchain details of the block number and gas used.

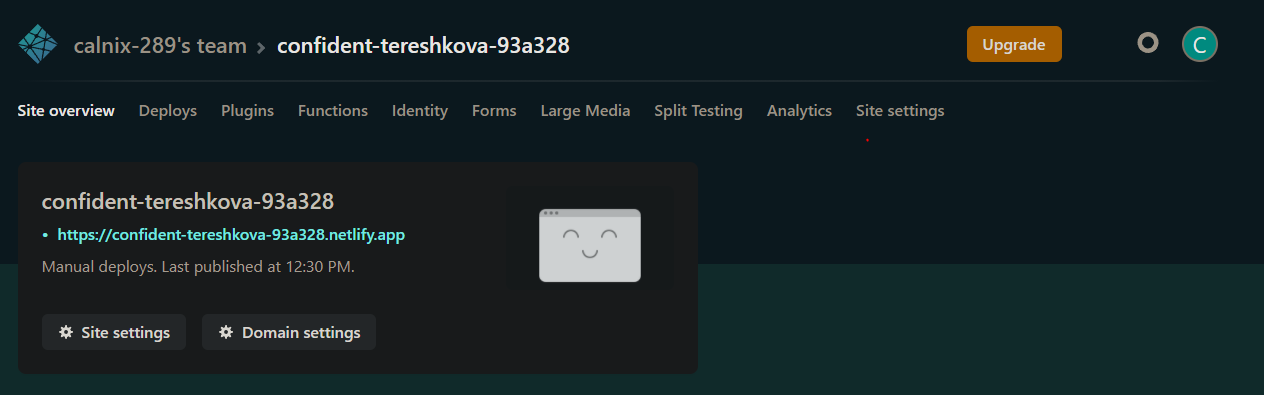
To access our file, we simply use the IPFS hash obtained as follows: https://gateway.ipfs.io/ipfs/<your IPFS address>

In this example, that would mean: <https://gateway.ipfs.io/ipfs/QmYFQKAWgf9shjNm5CetXz2r1vJ5FHd1FXaWh4votXNSN4> 

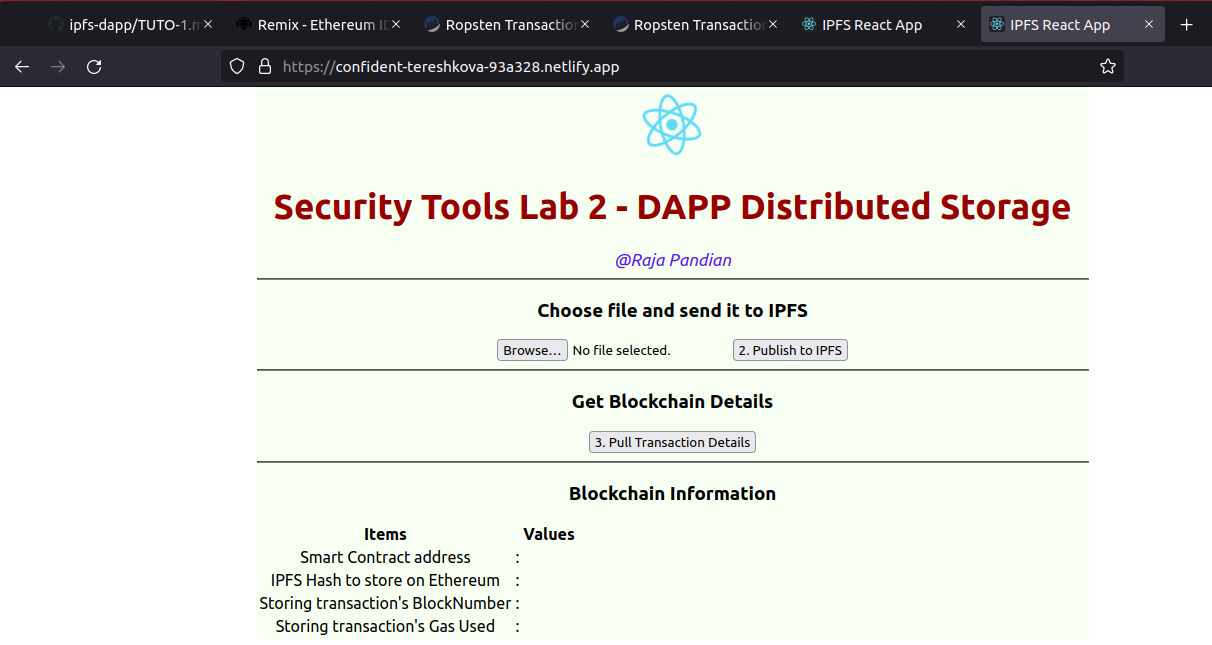
## Production Deployment

The blockchain backend is already deployed in production on Ropsten, however currently the frontend needs to be moved to production. To achieve this, we use Netlify, a free web application hosting service.

To build our production package, in our project root folder, we run npm run build. This creates a build folder which we will upload into Netlify.



The DApp can be accessed at <https://confident-tereshkova-93a328.netlify.app/>



1. At time of writing (06/10/2021) gas prices were at 90 Gwei (0.00000009 ETH) and ETH/USDT priced at US$3500. Please refer to <https://etherscan.io/gastracker> for updated prices. [↑](#footnote-ref-1)