CMOWE - A Blockchain CrowdFunding platform

A Project Report
submitted in partial fulfillment of the requirement for the degree
of
BACHELOR OF TECHNOLOGY
(Computer Science and Engineering)
To

By

Kuluru Vineeth Kumar Reddy	**18BCS043**
Karthick P S	**18BCS038**
B Ragavan	**18BCS016**
K Laxminarayana	**18BCS037**

Under the guidance of
Dr. Uma Seshadri
DEPARTMENT OF COMPUTER SCIENCE
IIIT,Dharwad

INDIAN INSTITUTE OF INFORMATION TECHNOLOGY DHARWAD
CERTIFICATE

This is to certify that the work contained in the project report titled Rise Together-A crowdfunding platform for Movies by Kuluru Vineeth Kumar Reddy, Karthick P S, Ragavan, K Laxminarayana was completed during the VII semester - IV Year as a Minor Project under the guidance of Dr. Uma Seshadri, IIIT Dharwad.

Signature of Supervisor
Dr. Uma Sheshadri
Professor, IIIT Dharwad

Declaration

We declare that this written submission represents my ideas in my own words and where other's ideas or words have been included, we have adequately cited and referenced the original sources. We also declare that we have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in our submission. We understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Kuluru Vineeth Kumar Reddy(18BCS043)
Karthick P S(18BCS038)
B Ragavan(18BCS016)
K LaxmiNarayana(18BCS037)

Approval Sheet

This project report entitled Rise Together - A crowdfunding platform for Movies by Kuluru Vineeth Kumar Reddy(18BCS043), Karthick PS(18BCS038), B.Ragavan(18BCS016), K Laxminarayana(18BCS037) of Indian Institute of Information Technology, Dharwad is approved for the degree of Bachelor of Technology in Computer Science and Engineering.

Supervisor
Dr. Uma Seshadri,
Professor,
Computer Science and Engineering,
IIIT Dharwad

Head of Department

Dr. Uma Sheshadri,
Professor
Computer Science and Engineering,
IIIT Dharwad

Examiners

Dr. Uma Seshadri,
Professor,
Computer Science and Engineering,
IIIT Dharwad

Table of Contents

1. Abstract

With the advent of blockchain technology, the film industry too is primed for disruption on a fundamental level. No longer are big-budget films greenlighted by a few Hollywood studios. By pre-selling movie tickets as ERC20 tokens, a vibrant marketplace becomes possible that brings global awareness and necessary funding to filmmakers of almost any background. As ERC20 compatible, such movie tickets are easily tradable and resellable and can be listed on any decentralized exchange. Potentially, we believe they can garner more investments than the hottest tech ICOs of late. There are over 7 billion people in the world. At one dollar each, isn't it rather obvious? More importantly, the economic incentive favors stories and storytelling that touch upon our common humanity.

Also with the power of Blockchain to act as a facilitator for the back end computations using smart contract deployments(to implement the business logic), it has become credible to establish complete transparency as well as consensus among the entities present. Public-key cryptography and IPFS hashing(to store large sized contents such as media files) have proved the ability of blockchains to replace the traditional way of computing(using databases) and is ought to be the digital transformation weapon of several industries. The film industry is one of such demonstrations here.

Blockchain provides a decentralized, transparent database that ensures trust between network participants. The participants can be partners in a supply chain or businesses in a consortium, who use blockchain to record. Blockchain permanently stores older blocks of data to maintain an unalterable series of records and transactions. This ensures that the provenance of every new block can be verified and traced back throughout the chain's history.Data integrity is a foundational property of data stored on the blockchain. Accessing and modifying the data stored on the blockchain is nearly impossible without notifying and seeking consensus from the entire network.
Traditional data storage and security systems are highly centralized, which means they have a single point of failure. This implies that any external attack on the central server like malware or brute-force hacking attempts may lead to partial or total loss of stored data.

These days we do see a lot of movies being released in piracy sites even before their official release dates. But with the advent of Blockchain technology we can use tokens that are valid only for certain hours and only. Also with the consensus mechanism the person who leaked the film is immediately caught as every transaction is noted in the distributed ledger after verifying with all the entities of Blockchain. **So Blockchain with the help of crowdfunding opens up immense opportunities for industries to raise funds in a transparent and efficient manner.**

2. Introduction

In today's world of creativity and innovation, the population is occupied with various business ideas. From the promoter's point of view, generation of business proposals are easier than arranging for the fund for its application. At present, the businessmen have many options from which they can arrange funds namely shares, banks, private lenders, venture capital institutions and other financial organizations. But still there are various frequent expenses which are not financed by these organizations. To deal with these initial capital and indirect expenses a new concept has arisen that is crowdfunding. Crowdfunding is widely used in developed countries like America whereas in India, the population is still less aware about it. The present research study is done with an intention to identify the development of crowdfunding in India. For this many research studies done in India and outside India have been referred.

As per IOSCO staff working paper, 2014, crowd funding can be segregated into four categories, namely **donation based crowdfunding**, **reward based crowdfunding**, **peer to peer lending** and **equity based crowdfunding**.

1. **Equity-based Crowdfunding** is asking a crowd to donate to your business or project in exchange for equity.
2. **Donation-based Crowdfunding** is asking a crowd to donate to your project in exchange for tangible, non-monetary rewards such as an e-card, t-shirt, pre-released CD, or the finished product.
3. **Debt-based Crowdfunding/peer to peer lending** is asking a crowd to donate to your business or business project in exchange for financial return and/or interest at a future date.
4. **Reward-based crowdfunding** refers to the process of solicitation of funds in which investors earn some existing or future tangible benefits as return on their investment.

According to Crowdsourcing.org 2015, in 2014 1,250 crowdfunding platforms raised total capital of $16.2 billion. At that time North America was considered as the largest market with rapid growth in Asia, particularly in peer to peer lending.

Funding in reward-based crowdfunding projects has grown rapidly in the past decade. In 2014, the share of newly created platforms that are reward-based was approximately 40%, followed by each, at around 20%, in donation-based and lending-based platforms (Belleflamme et al., 2015). The present study focuses on reward-based crowdfunding as it is the largest crowdfunding type in terms of the overall number of crowdfunding platforms as well as the funding amount raised being the fastest growing form of crowdfunding. In addition, we can explore supporter's behavior and motivation for their decision making in reward-based crowdfunding as its nature is similar to the process of customers making purchase decisions.

Macht and Weatherston (2014) defined crowdfunding as a new fundraising tool for small business projects. This is profitable for fund-seeking companies by reducing the fundraising problems, providing entrepreneurs other value added benefits like publicity, contacts and increasing access to further fundraising. The most important thing is there is no loss of control and ownership.

Schwienbacher and Larralde (2010) explained crowdfunding as an open call to the mass of population through the internet, for the requirement of financial resources whether in form of donation or in compensation of some reward or any voting rights in order to support initiatives for specific purposes.

Many researchers were of the view that crowdfunding is closely related with many of modern financial innovations like micro credit, crowd sourcing etc. They also thought of crowdfunding not only as a source to generate the financial assets for startups and enterprises but also as a means of building a customer base and developing mutual trust and transparency with the crowd.

Some of the popular platforms that have demonstrated success in crowdfunding are:

1. **Kickstarter-** a platform that focuses on creative projects like art, music, film, etc. Funding is all or nothing and fees are fairly reasonable.
2. **Indiegogo**- a clear choice for best overall for its track record of success in helping to fund more than 800,000 ideas all over the world since 2008.
3. **Mighty Cause**- a fundraising platform to help nonprofits and individuals raise money online for their causes. Its flexibility for fundraising gives it the edge in this category for organizations seeking donations, as it has helped more than 150,000 causes attract the contributions they needed to meet their goals.
4. **StartEngine** - a standout option for those looking to invest some money because the platform allows everyday people like you to put your money directly into companies and startups you admire and get equity in return, for as little as $100 to start.
5. **SeedInvest Technology:** A platform that has attracted 500,000 investors and helped over 235 startups raise more than $300 million in financial backing, making it our choice as best for startups.

Crowdfunding worldwide has considerably grown over the years and will continue to do so in the foreseeable future. After all, it presents a viable means of raising significant amounts of funds coming from backers around the world. In fact, the latest general crowdfunding statistics reveal that more funders will come from the APAC region in the coming years. And the types of products or businesses funded are varied, be it video games, writing, food, or fashion.

The global crowdfunding market size was around $30 billion in 2019 with an annual growth of more than 16% from 2021 to 2026 (Mordor Intelligence, 2020). Crowdfunding added $65 billion to the worldwide economy. There are 1,478 crowdfunding organizations in the US. Moreover, 6,445,080 crowdfunding campaigns were held in 2019 with an average crowdfunding donation of $99.

Hence from the above figures it is quite evident that the potential of crowdfunding as a sector to build trust and generate huge capital along with a good customer base is something that cannot be avoided. When such crowdfunding campaigns are able to generate huge revenue in the European and American market, why not implement them in the Indian market(with a higher population and such good potential for varied sectors such as arts, film making, etc.).

3. Problems encountered and solution proposal

After complete brainstorming and thorough research from various sources, it was observed that the film industry in India is operating in a complete manual process which has the scope of digitization using technology. Also there is very little concentration on the post production process where distributors are facing a huge loss due to the current system which is not transparent, hence resulting in problems such as suicides,etc.

Hence, after all the above observations,a decision has been made to focus on the below mentioned major issues that the film industry in India is facing:

1. **Title registration process**: The current system of film industry uses a manual process of title registration which needs a film director to physically go to a Film Producers Association and register his title. Then the association reviews the validity of the title and issues an acceptance to register the specific title(if unique and legitimate).

2. **Facility to sell rights of the movie:** With the current system of film rights distribution from a regional level to a global level(for example remaking a Telugu film into a Tamil film), the mutual agreement between the directors/producers happens physically by signing(Here also forgery signatures are a threat).

3. **Problems faced by the Distributors of the film**: There is a general misunderstanding that the entire profit/loss of a film is borne by the producer of the film. But, the reality is that the profit/loss is completely borne by the distributors of the film who buy a film at a given amount from the producers and sell the reels/movie to the respective theatres. Now depending on the no. of screens in the theatre, the percentage of the amount called distributor share is given to the distributor(usually 90% for single screens and 50% for multiplex screens). Here the problem is that the distributor is unable to choose the proper film(based on director, theatre count allotted for film,etc.) to make as much profit as possible. Due to the lack of such a feature/functionality for forecasting in current film industry, lots of suicide cases have been reported in recent years and similar trend is continuing.

4. **Hurdles to Independent Film Makers:** With the days rolling out the talented and passionate content creators coming out on to the board to showcase their vision on screens is dramatically increasing,but to vain they are facing unavoidable obstacles to procure enough funds from well known producers and struggling real hard to turn their paperwork to pictures on screen.

Proposed solutions for the above mentioned problems:

1. **In the title registration process:**

   A **blockchain based platform** could be used as a solution to the above stated problem by using a distributed ledger for recording the entire set of transactions, which are movie titles in our case where a business logic that no two directors can register the same title can be introduced with the help of smart contracts.

2. **In facilitating to sell the rights of the movie:**

The second problem could be solved with the help of **Blockchain as well by using tokens**(such as ERC20 tokens) to enable the exchange of digital assets(for example in our case the agreement between the directors/producers to sell the movie remake rights) that can be sold/bought by the respective entities, all of whose details are also again stored in the distributed ledger of blockchain and also the transactions are verified by the various entities in the blockchain where possibility of a forgery is completely eliminated.

3. **In problems faced by Distributors Case:**

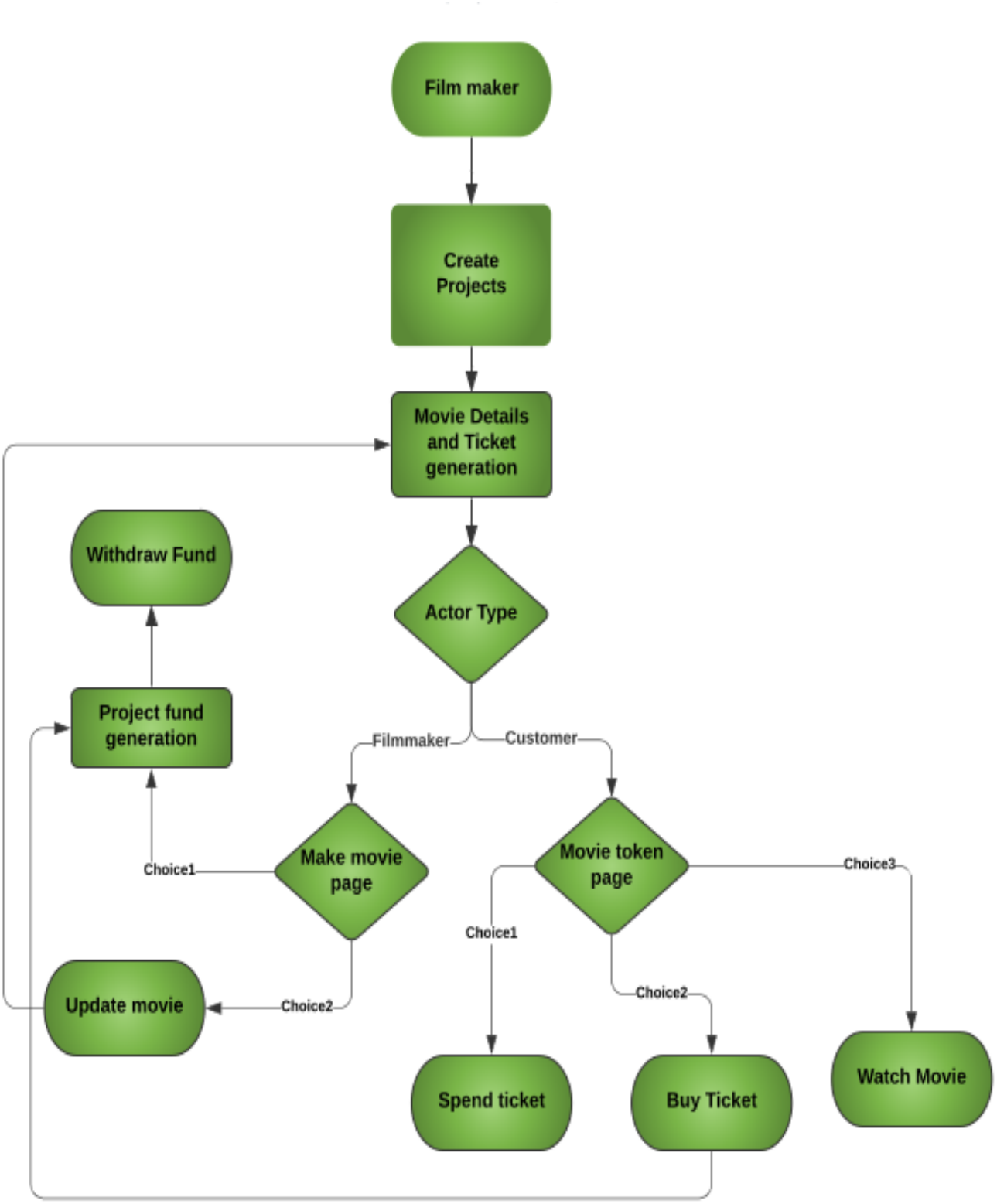
Now the third problem that we encountered, can be resolved by forecasting techniques using **machine learning** for predicting the approximate profit that a distributor might get by buying the distribution rights of a film from the producer based on the features like the history of success rate of the director(which can be measured based on his no. of previous hits and flops), the actors involved in the film,etc. Based on which he can choose the best movie to select to attain maximum profit. Also another possibility is to forecast the theatres that the distributors would have to choose to sell the movie reels to in order to get maximum benefit(based on the features like location of the theatre, no. of screens per day, type of screens allowed for the film,etc.)

4. **Innovative Solution for small FilmMakers:**

The fourth problem can be resolved by using a **crowdfunding technique** with the help of underlying blockchain technology by using the concept of **smart contracts** from normal people like us with similar interests in film related stuff or content lovers which helps an independent filmmaker to request for funds from the general audience(usually movie lovers).

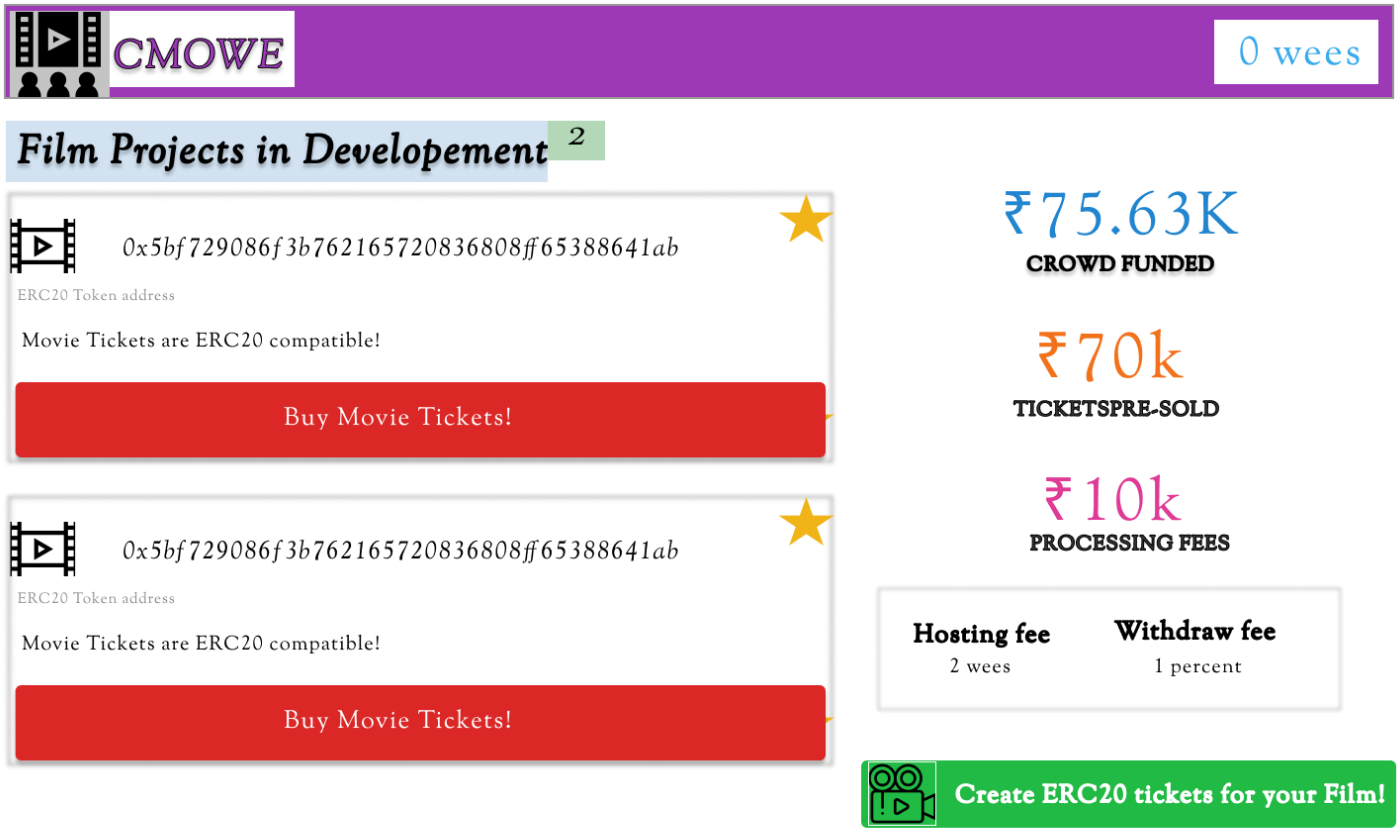
As a part of this project we will be exploring the solutions to points <u>1 and 4</u> using Blockchain technology where the crowdfunding aspects of making our crowd/viewers to invest in their favourite movies to get in return a movie ticket as a symbol of Reward for their contribution in the project has been taken care of.

Project Flow Diagram

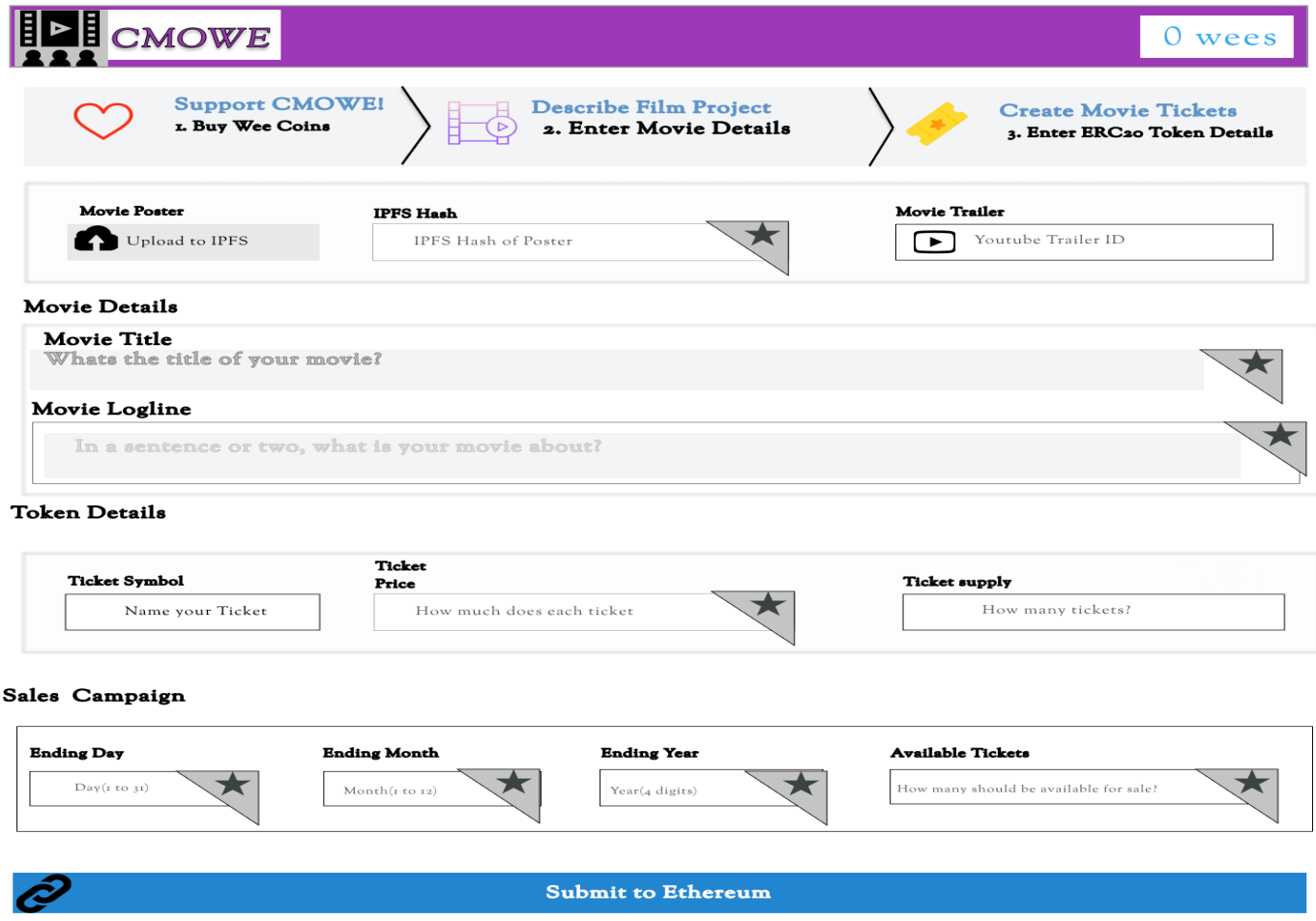


Design of our Blockchain Platform using Figma

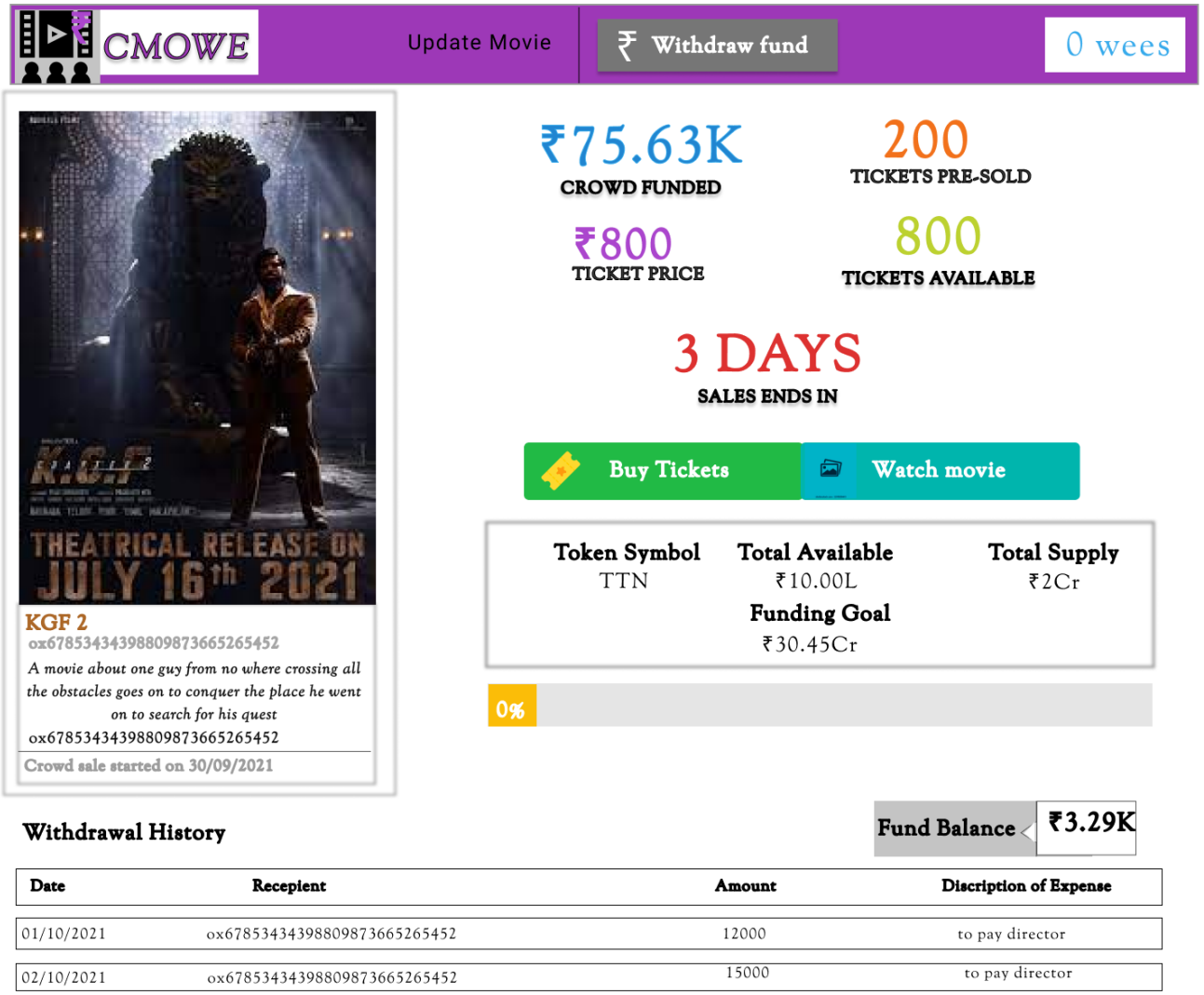
1. Studio Home Page



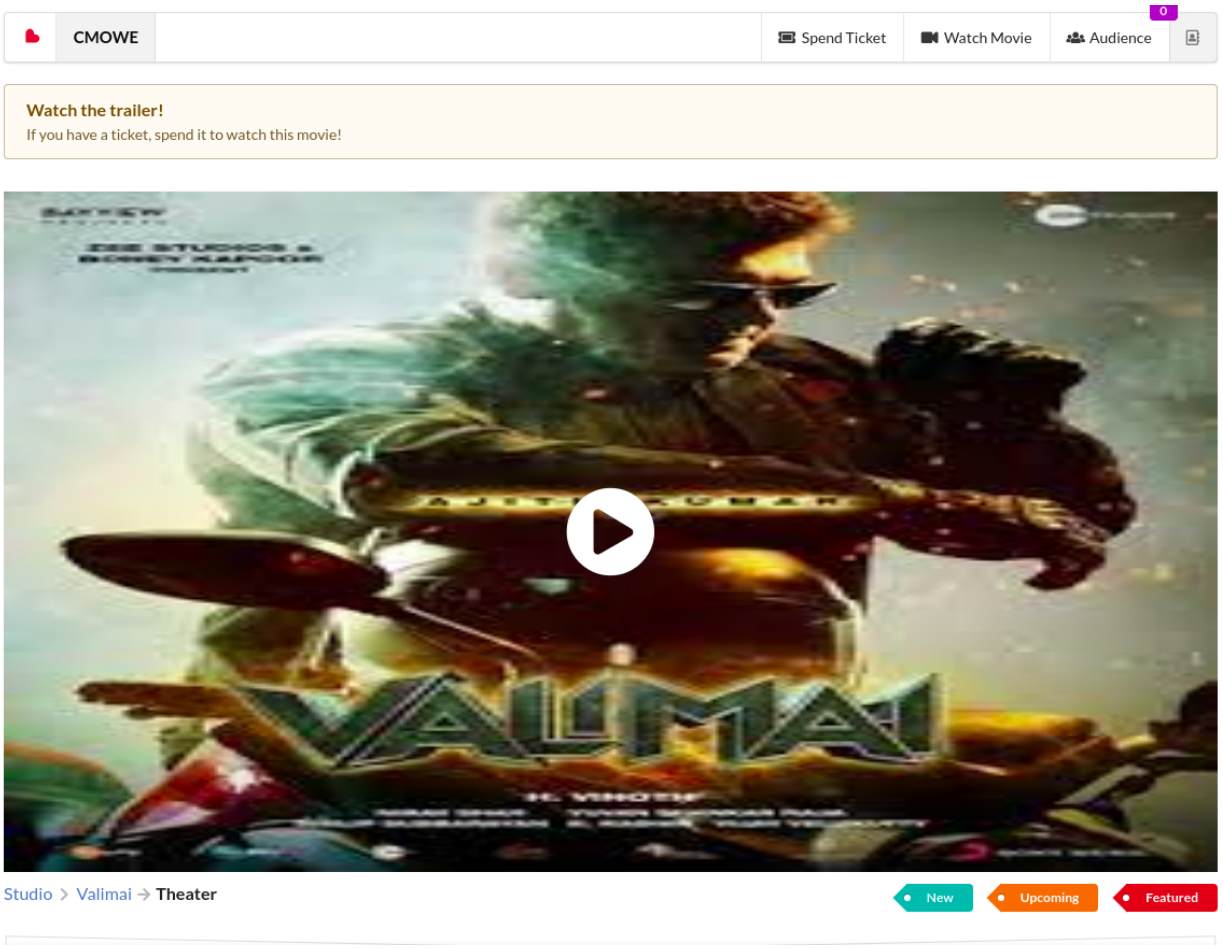
2.Make Movie Page



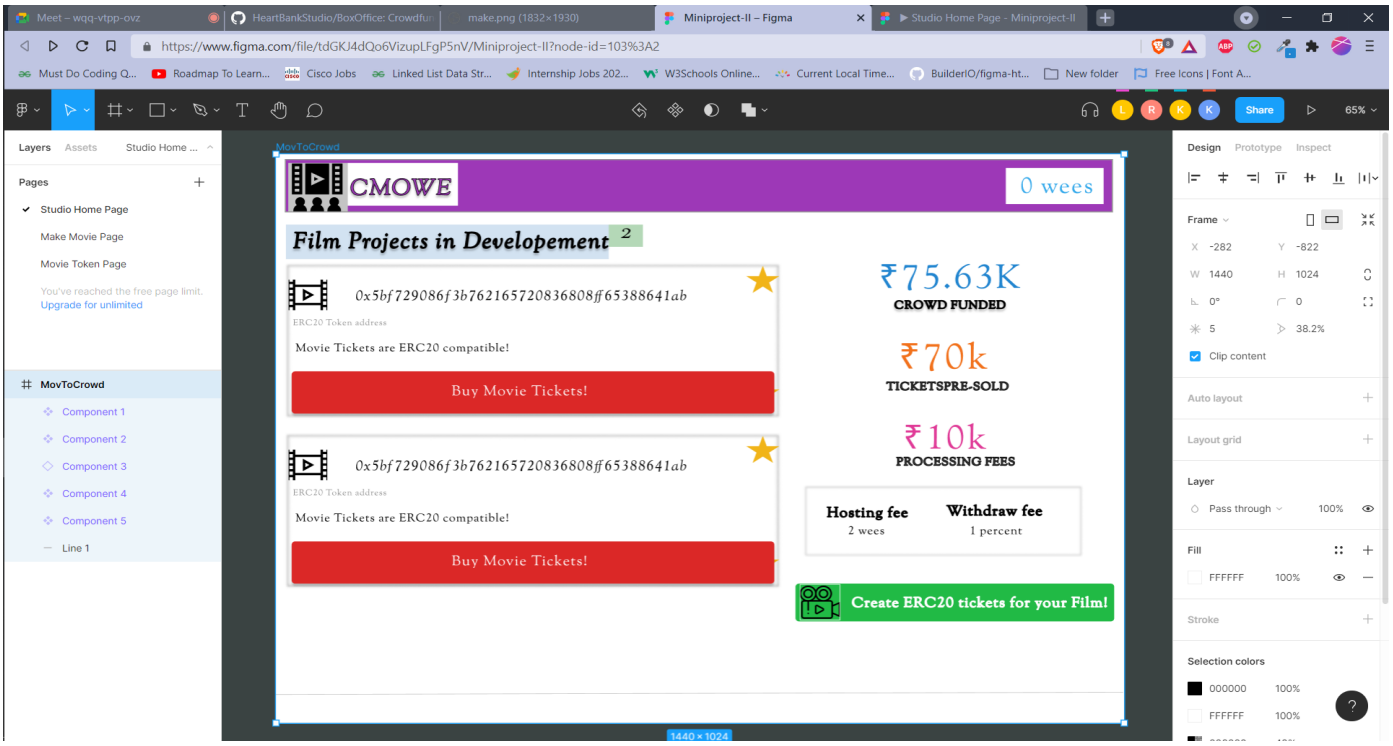
3. Movie Token Page

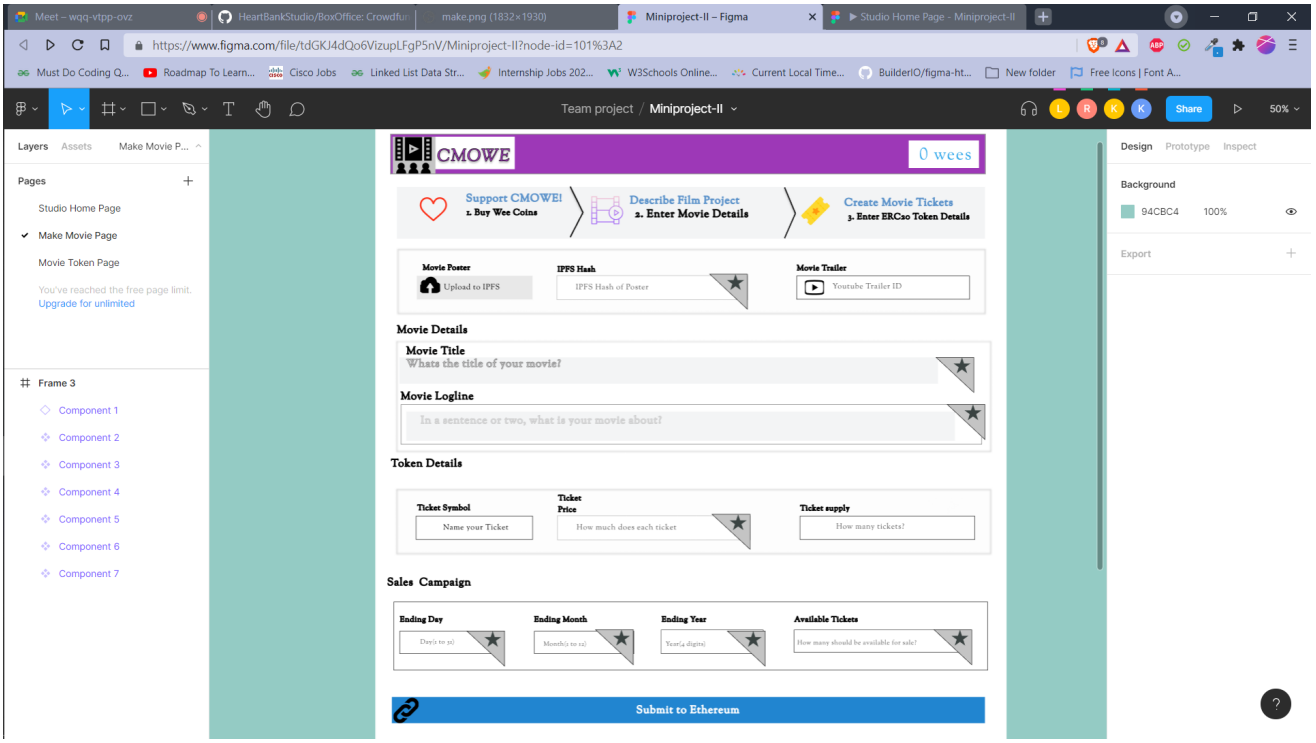


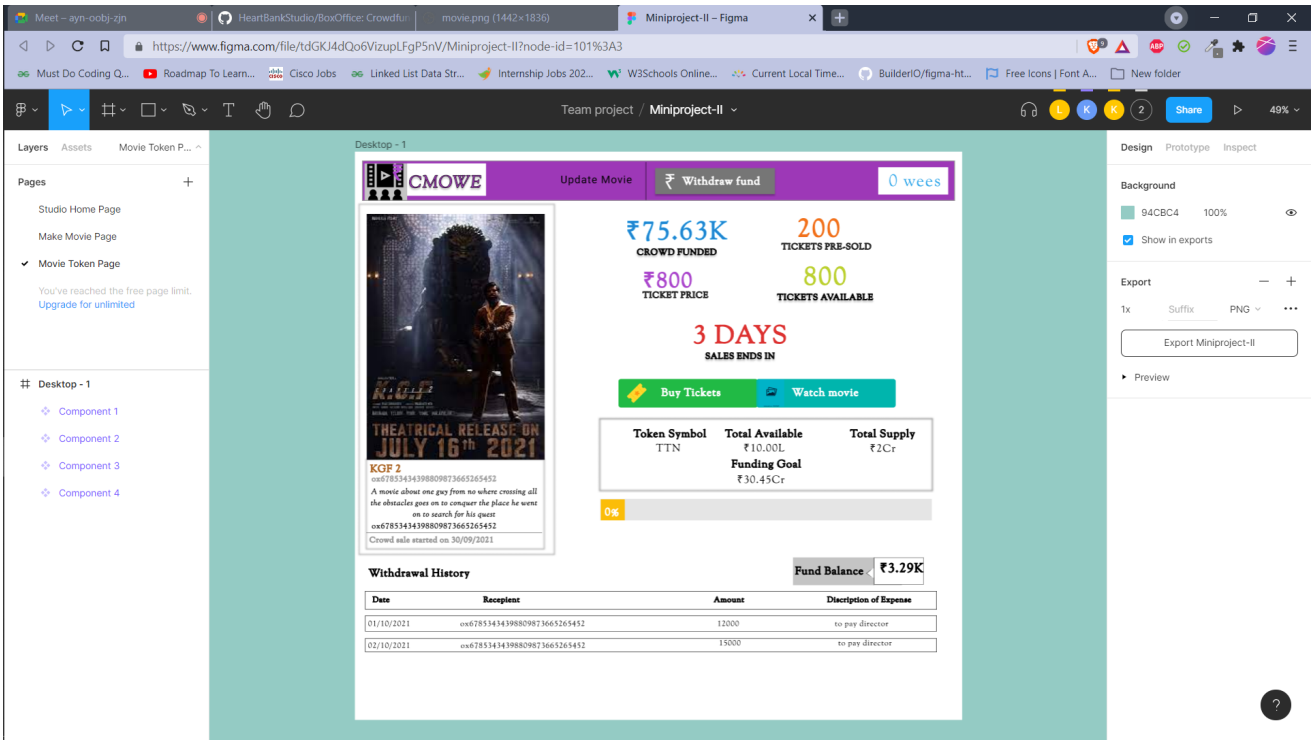
4. Watch Movie Page



Screenshots of Designs working in Figma







With the help of tools and technologies like Truffle, Ganache, Ethereum Virtual Machine(EVM), Metamask and web3js it was possible to demonstrate the backend computations by applying our

business logic. Truffle was used for the Smart contracts compilation , Ganache for having a local test blockchain running(to enable transactions), EVM for storing all our transactions and blocks(our actual ledger), Metamask for having a wallet to make transactions and web3js to make our backend(Blockchain) interact with frontend and deliver the queries accordingly.

Project solution proposal video

4. Technology Stack and Dependencies for building the Platform

1. **Front end:** As with most other web based platforms this project to be built using Blockchain requires the knowledge of a front end framework. For the prototype we would be using a **React framework** as it is completely coded in javascript and is relatively easier to understand and code. We also use the node and npm(with the compatible versions).

2. **Truffle:** A world-class development environment, testing framework and asset pipeline for blockchains using the Ethereum Virtual Machine (EVM). We require truffle for compiling our smart contracts before they are actually deployed in the blockchain and also to test our smart contracts and deploy them into the blockchain.

3. **Metamask:** MetaMask allows users to store and manage account keys, broadcast transactions, send and receive Ethereum-based cryptocurrencies and tokens, and securely connect to decentralized applications through a compatible web browser or the mobile app's built-in browser. We use Metamask by installing it from chrome plugins that help our entities/actors in Blockchain to sign and send transactions.

4. **Ganache:** Ganache is a personal blockchain for rapid Ethereum and Corda distributed application development. We can use Ganache across the entire development cycle; enabling us to develop, deploy, and test your dApps in a safe and deterministic environment.

5. **Web3js:** web3. js is a collection of libraries which allow us to interact with a local or remote ethereum node, using a HTTP or IPC connection. The web3 JavaScript library interacts with the Ethereum blockchain.

6. **Infura:** Infura provides the tools and infrastructure that allow developers to easily take their blockchain application from testing to scaled deployment - with simple, reliable access to Ethereum and IPFS. For our platform we use this to store our videos(for movie trailers) into the IPFS by storing only the hash of the video and not the complete one into the blockchain. **We also used infura to deploy our solution proposal video to enhance our reach!!**

5. Architectural flow of our Blockchain model

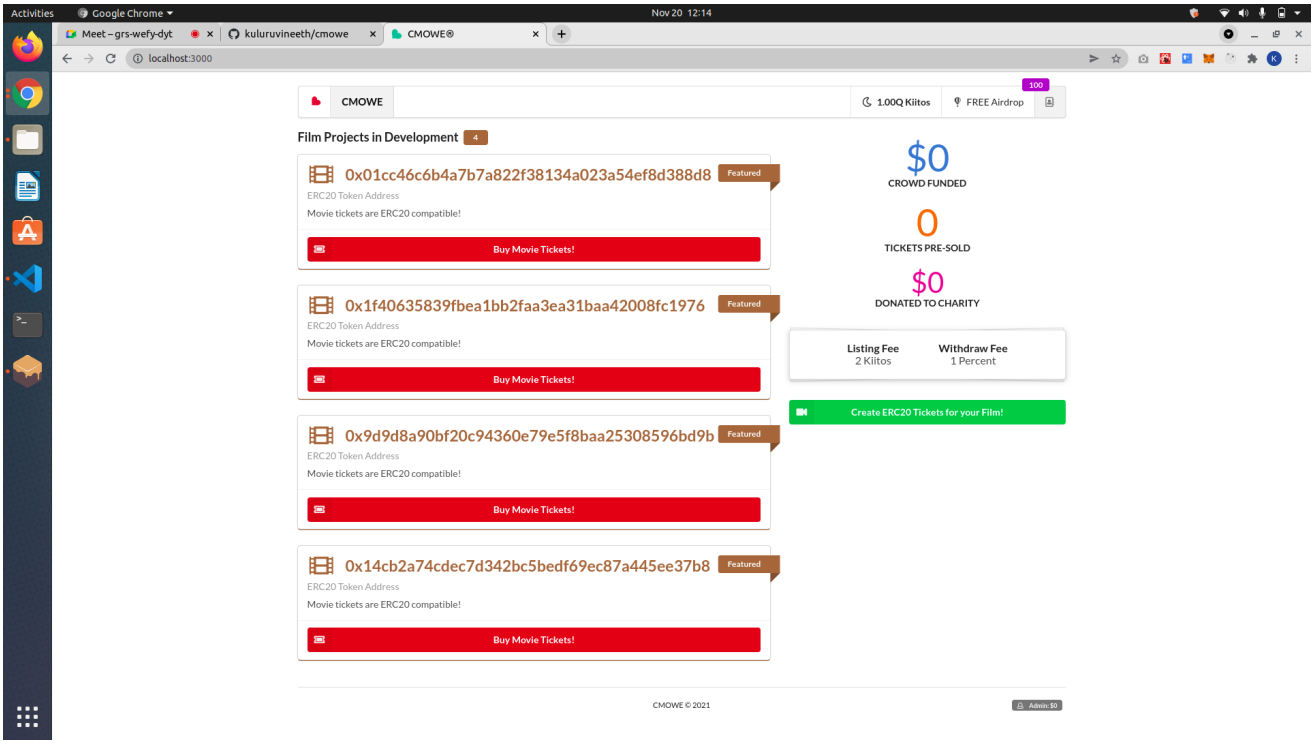
- Refer pg. 10 (Project Flow diagram)

6. Platform Features and Functionalities

There are four main pages in our platform, each mapped to corresponding URL routes for easy bookmarking and search engine optimization. Let's review each one in turn.
We use Kiitos coins which are ERC20 tokens that are used to purchase our movie tickets(which are exchanged for ethers using the coinbase which says the price of an ether in USD).

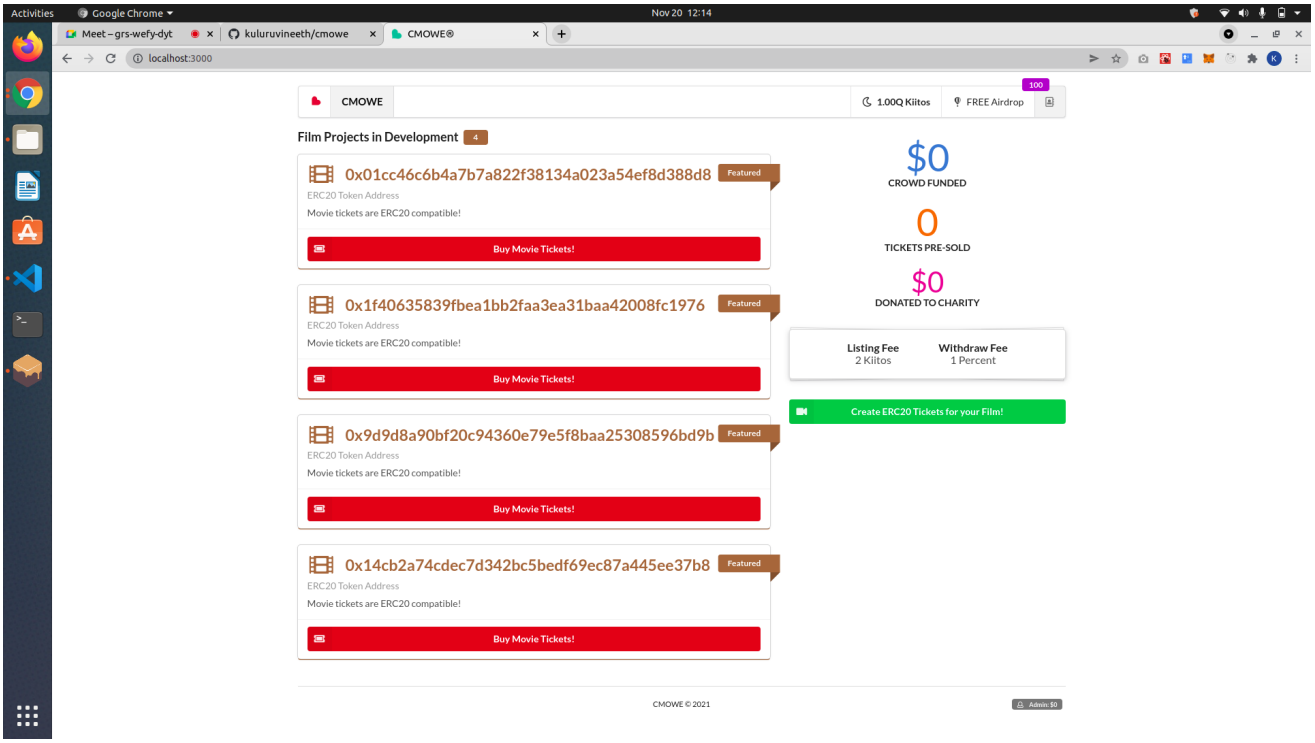
6.1 Studio Home Page

The studio page is the home page, the first page the user sees, and has the following functionalities:

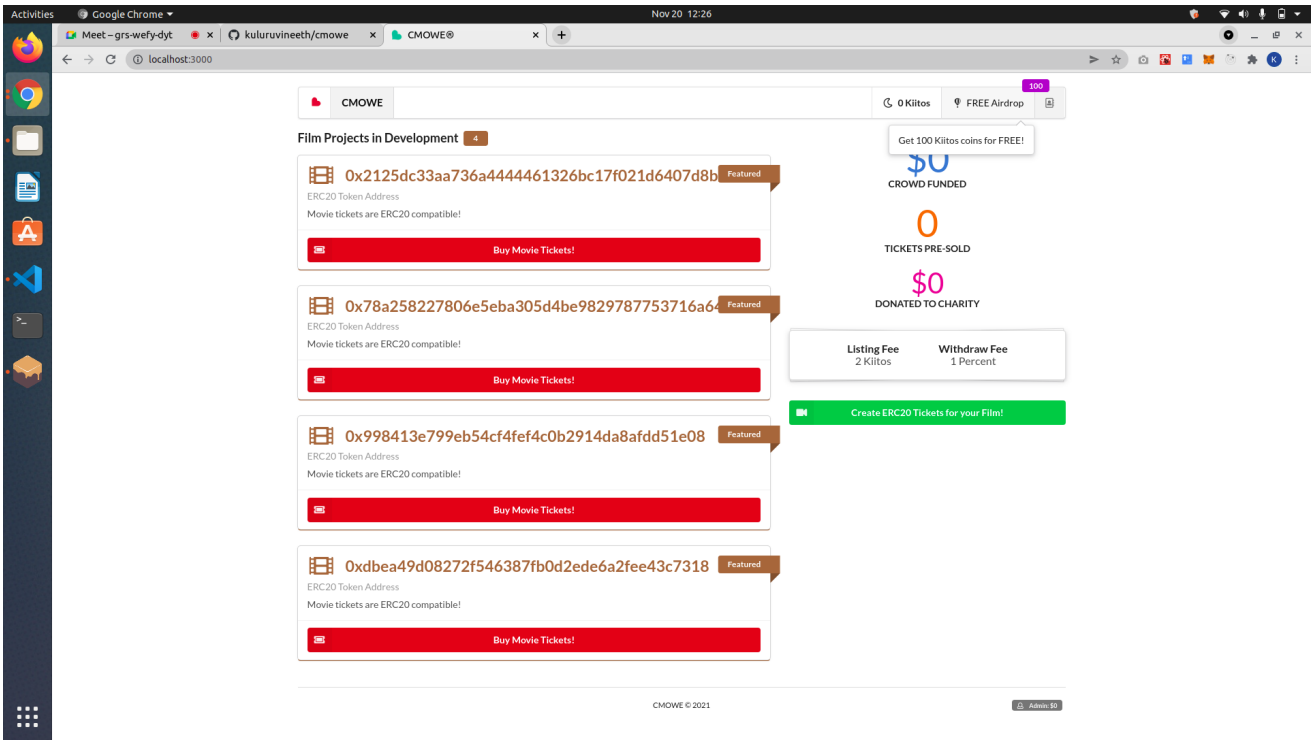


As an Ethereum application, Metamask is necessary for transaction signing. Therefore, this dApp checks for the presence of the web3 object injected by the Metamask extension, showing a warning message if undefined.

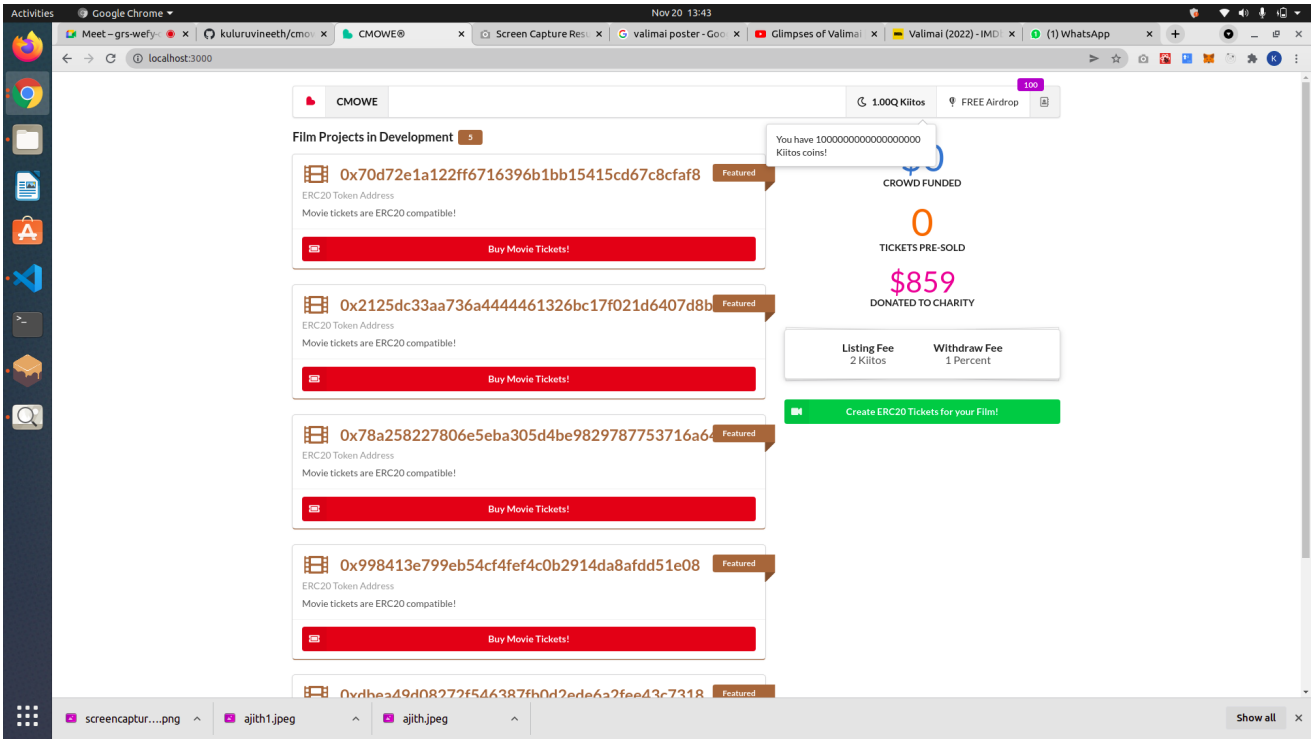
There are two main sections. The left shows all the currently active film projects in development, while the right displays encouraging box office stats. Large numbers are shorthanded for aesthetic appeal, but precise numbers can be viewed on mouseover.



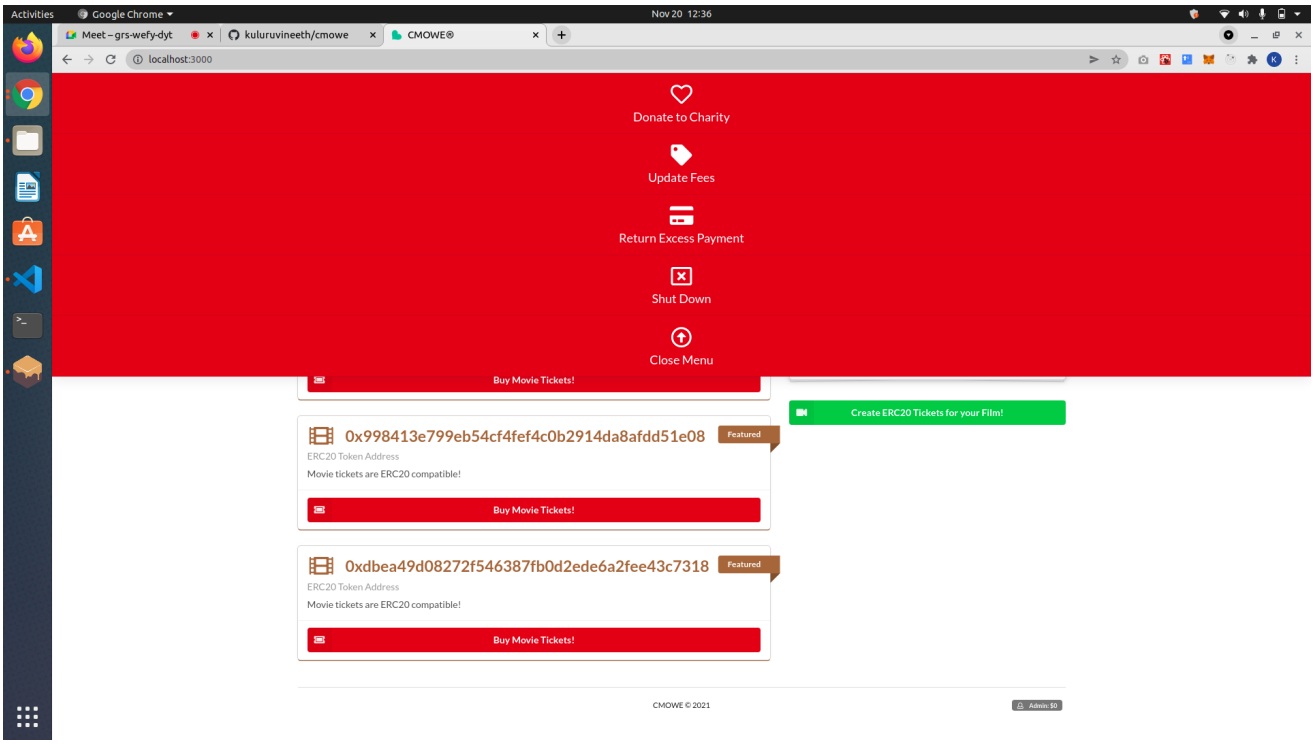
On mouseover, the currently selected Metamask account can be viewed as a tooltip to give the user confidence that it has properly connected with the dApp.



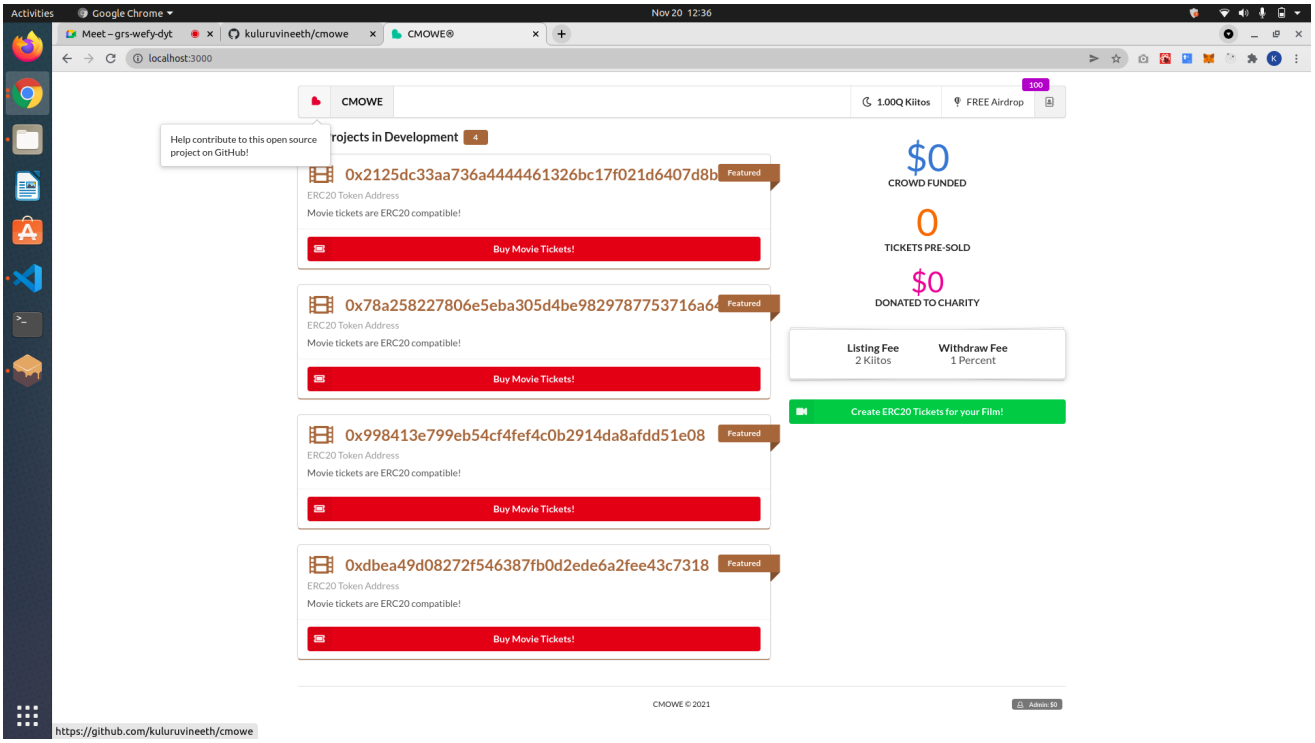
Two Kiitos coins are required to create a film project. For the filmmaker's convenience, 100 can be airdropped for free with the click of a button!



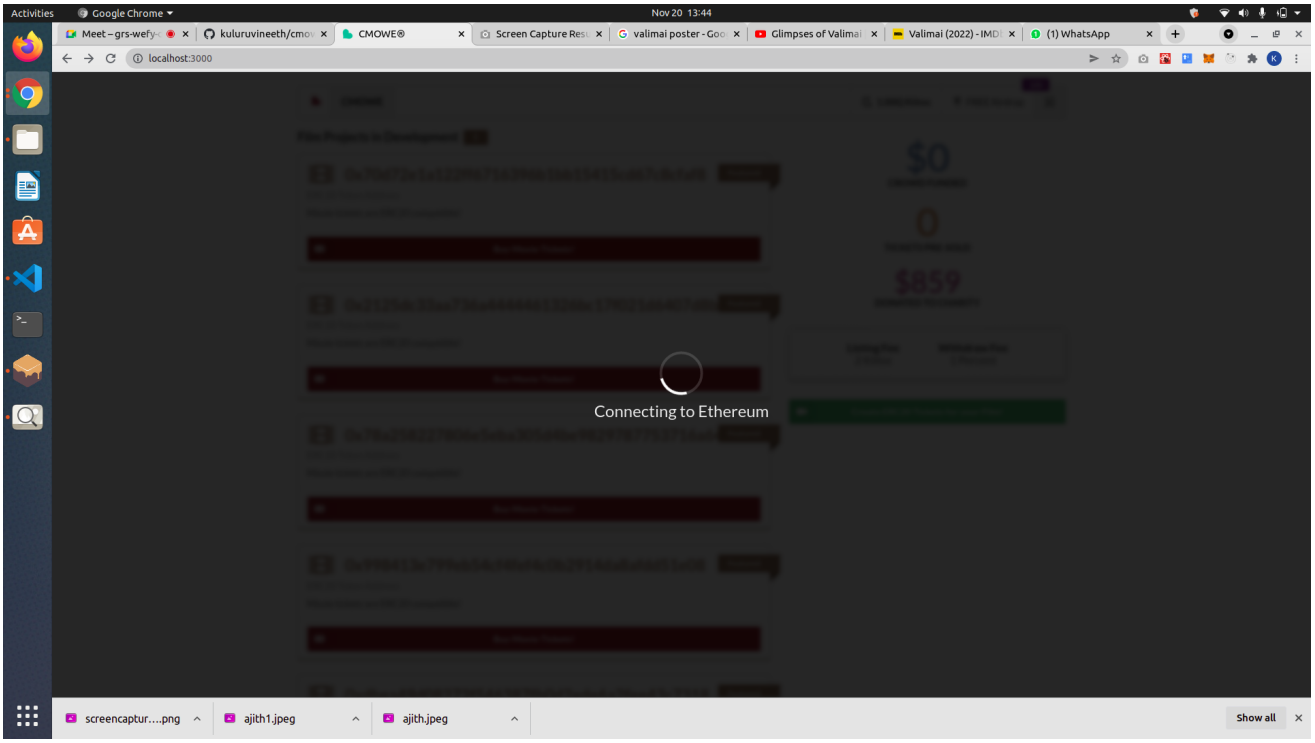
Also, the current balance of Kiitos is prominently shown. If it has been shorthanded, the exact number can be viewed on mouseover.



If the currently selected Metamask account is determined to be the admin, a special button appears on the bottom right corner of every page. Beside it, fees collected for charity (minus withdrawals) are shown rounded to the nearest USD. The exact ether can be viewed on mouseover. Clicking it drops down a menu with actions that the admin can perform.



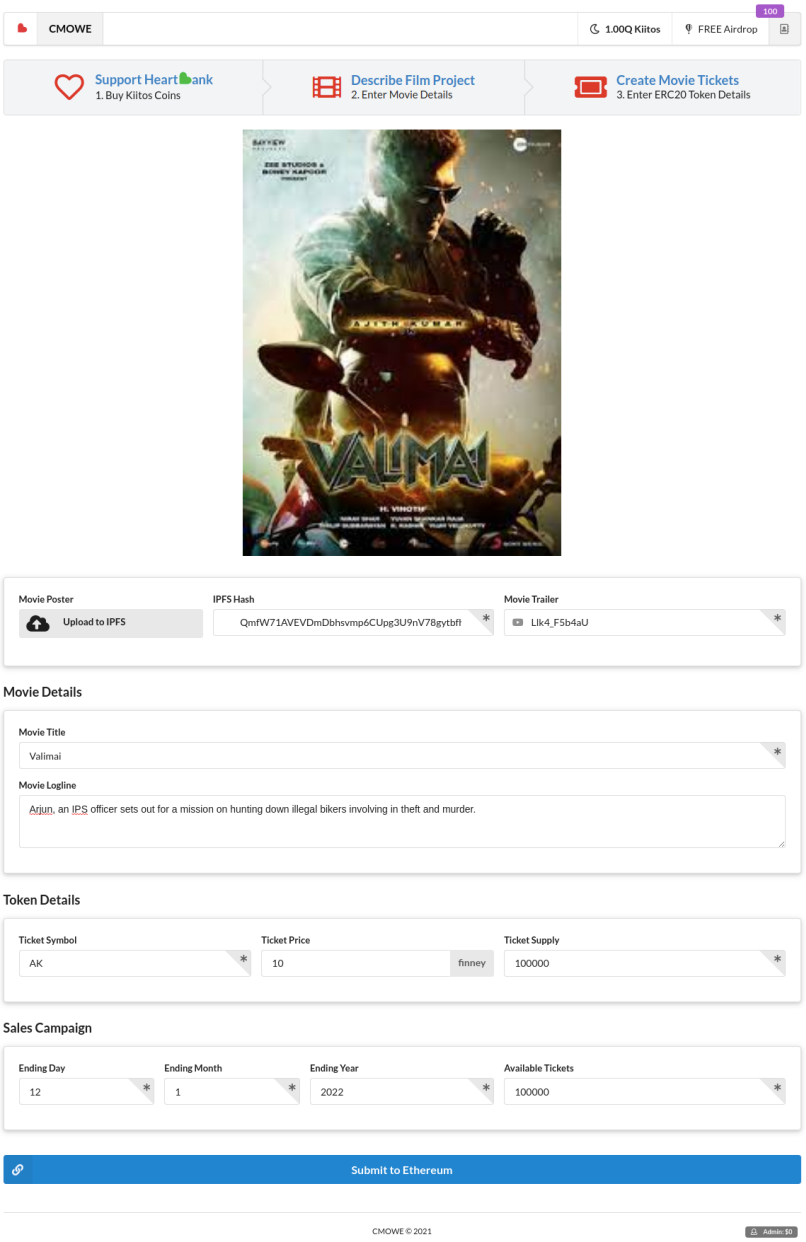
To attract open source contributors to this project, a button to the GitHub repo is marketed to all users.



Because retrieving contract data from Ethereum can take some time, a spinner is shown while the page loads.

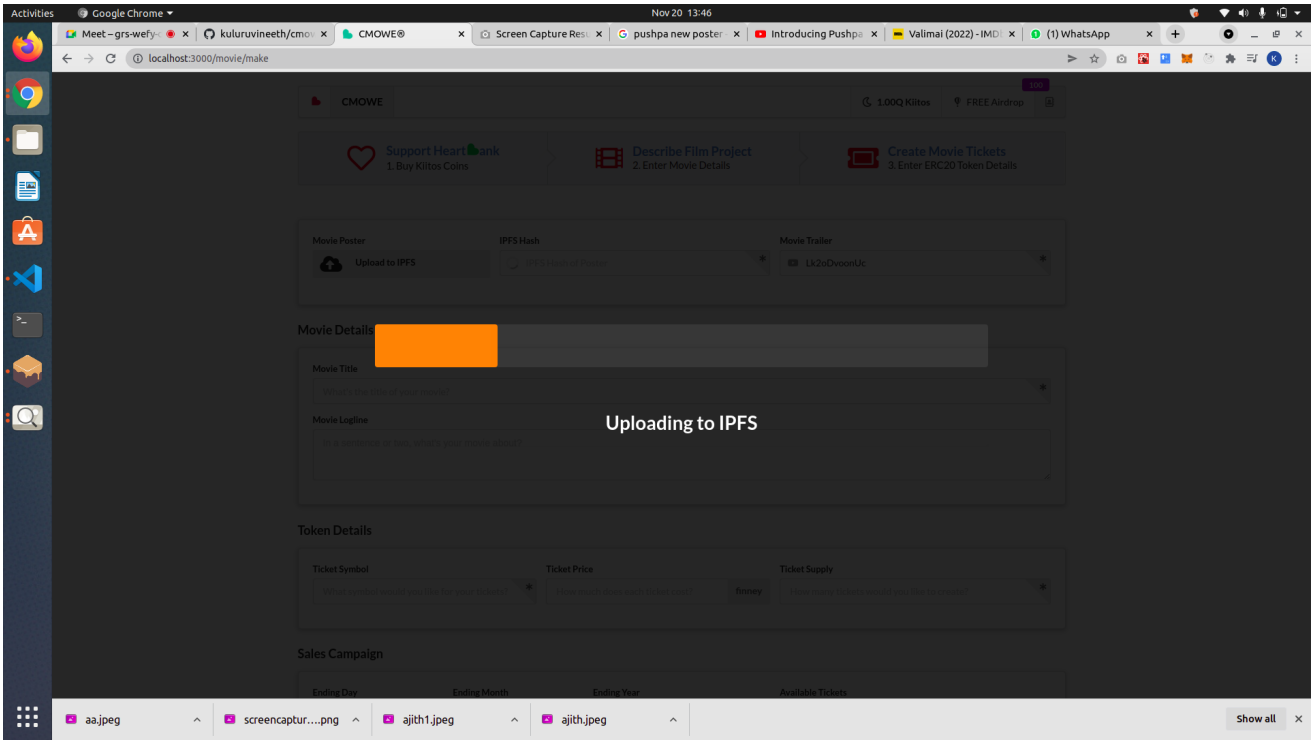
6.2 Make Movie Page

Here, filmmakers of any background can customize movie details to create ERC20 compatible tickets that are unique to each film. On the blockchain, this means a standard ERC20 smart contract is created for each project.

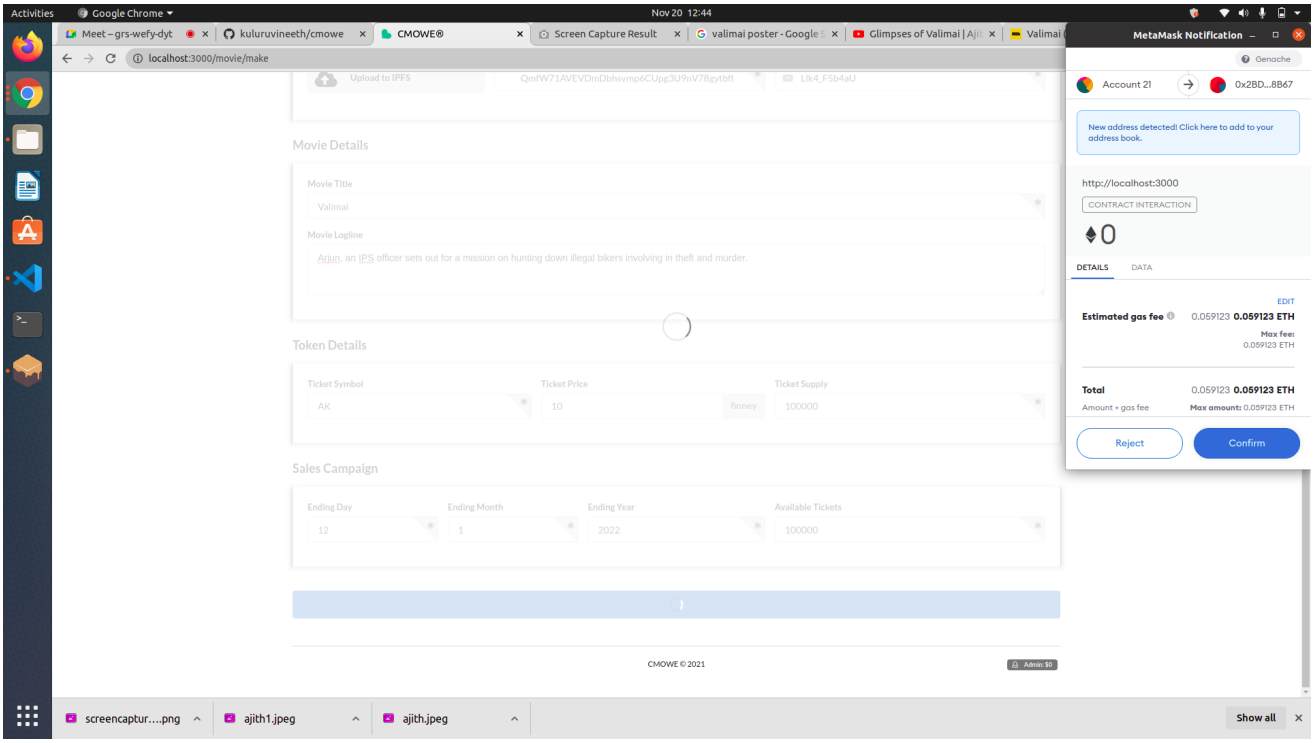


The overall process to create a film project and movie tickets is highlighted at the top. Since there are quite a few inputs that are needed from the filmmaker, they are sectioned into four categories for easier understanding. The **first section** asks the filmmaker to upload the movie poster to IPFS. If one already exists, the filmmaker can paste the IPFS hash in the provided field. This section also asks the filmmaker for the YouTube ID of the movie trailer. The **second section** asks the filmmaker for the details of the movie itself. Here, the filmmaker can specify the title and the logline. The **third section** asks for information related to the creation of a ERC20 contract. Here, the filmmaker can specify the

ticket symbol, ticket price, and total supply of tickets. In the **final section**, the filmmaker can provide the requirements of a sales campaign. She can specify the ending date in day, month, and year, and the available number of tickets during the sales period.



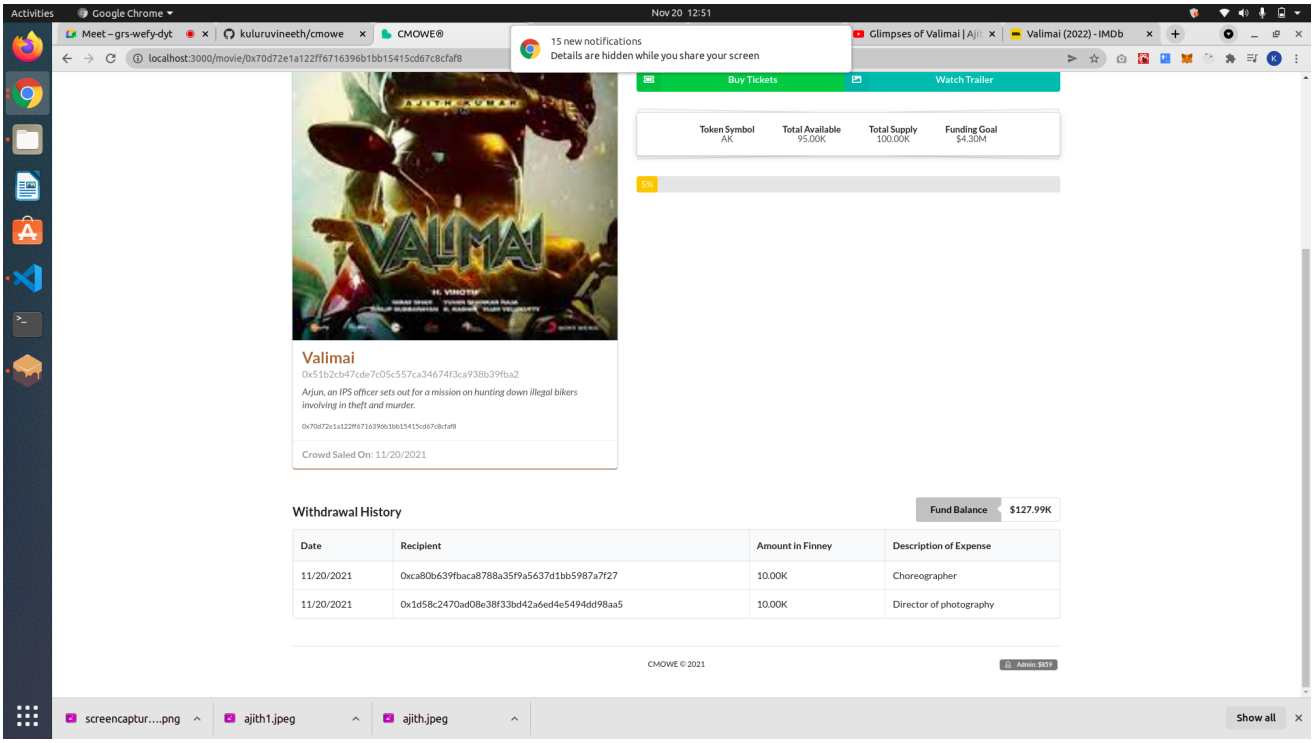
Uploading a high-quality movie poster to IPFS can take some time, so a progress bar is shown to give the filmmaker confidence that everything is okay. When finished, the hash that's returned is populated in its corresponding field in case the filmmaker wants to save it for future reference.
Also after success, a preview of the movie poster is centered on the page to visually encourage the filmmaker to finish filling in the remaining details!
While the filmmaker confirms and signs the transaction with Metamask, the form is disabled and a **spinner** is shown. In fact, the spinner is a must because transactions can take a while to get processed by the network.



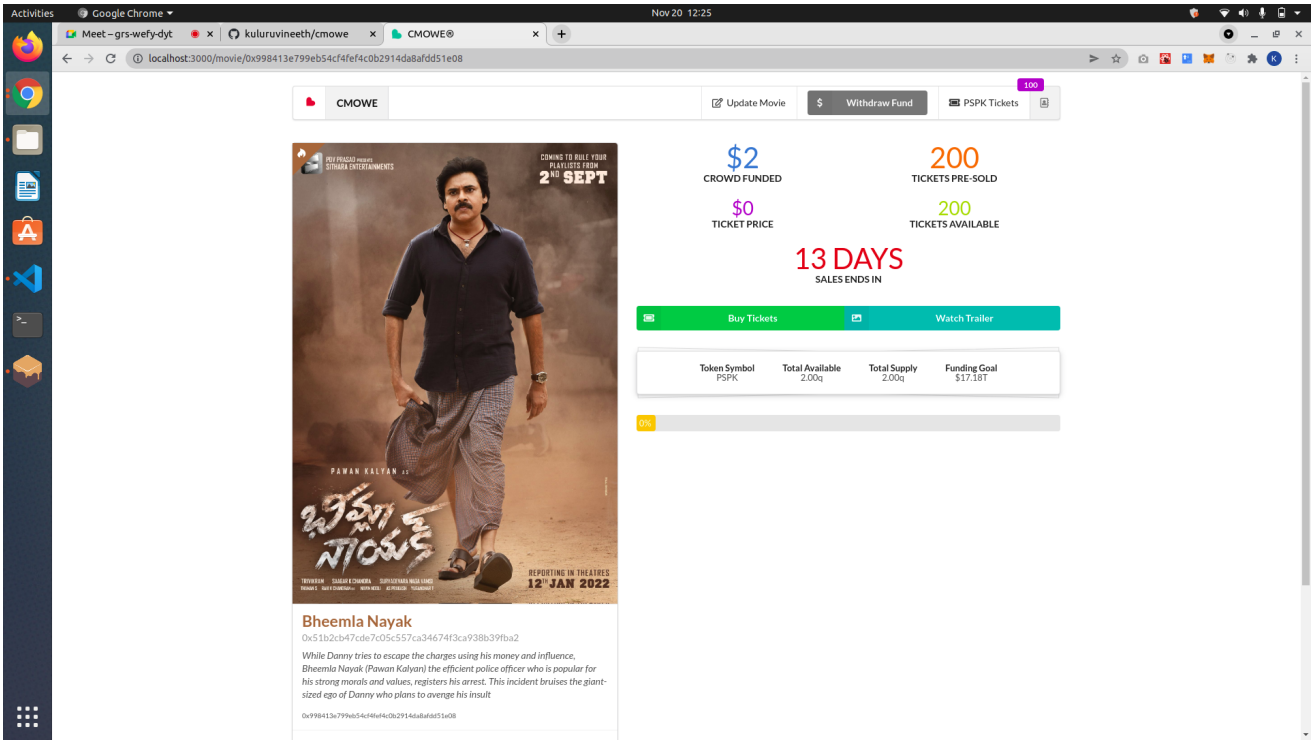
Metamask is awesome because it performs a dry run of the function call to calculate the necessary gas fee and return any thrown exception automatically for the user's convenience.If Metamask throws, the error message is passed back to the dApp to inform the filmmaker of corrections needed.After completion, the filmmaker can return back to this form to make changes (except the total supply of tickets). Input fields are pre-populated with prior data for convenience.

6.3 Movie Token Page

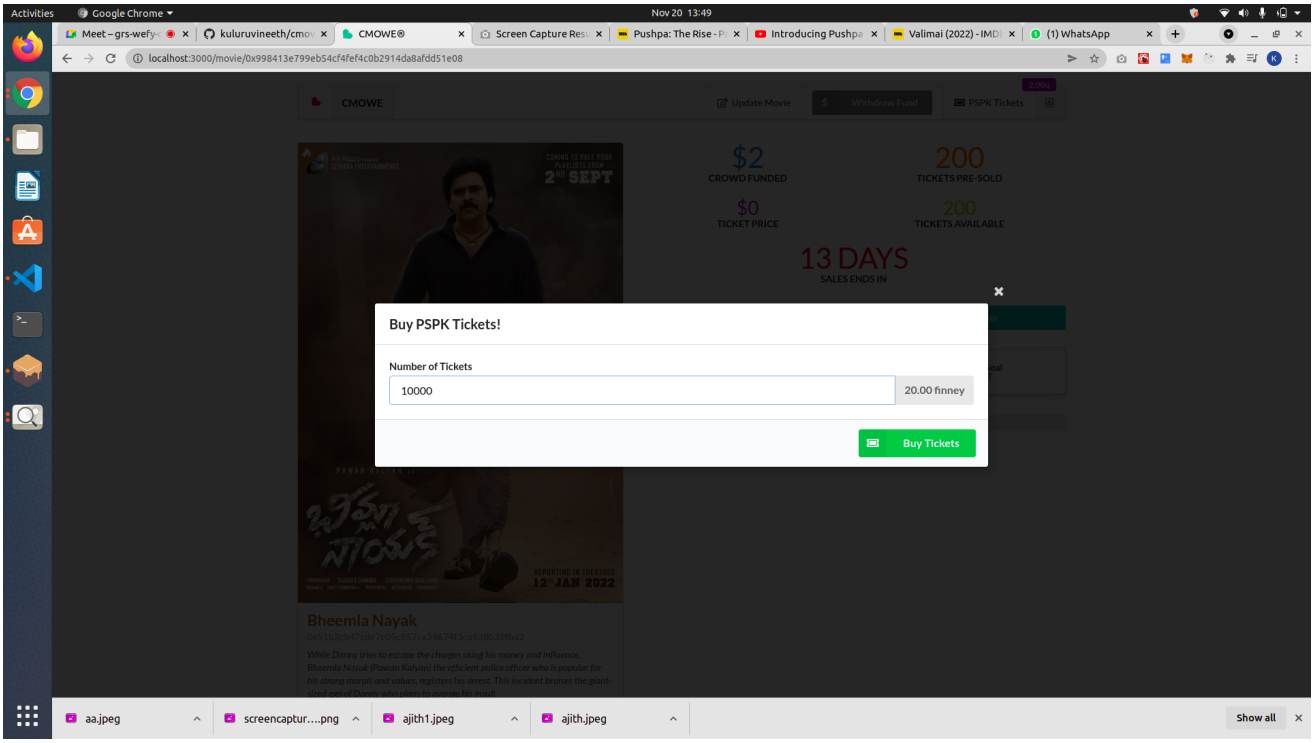
After successful creation, the public can visit this page to learn about a film project and pre-buy tickets.



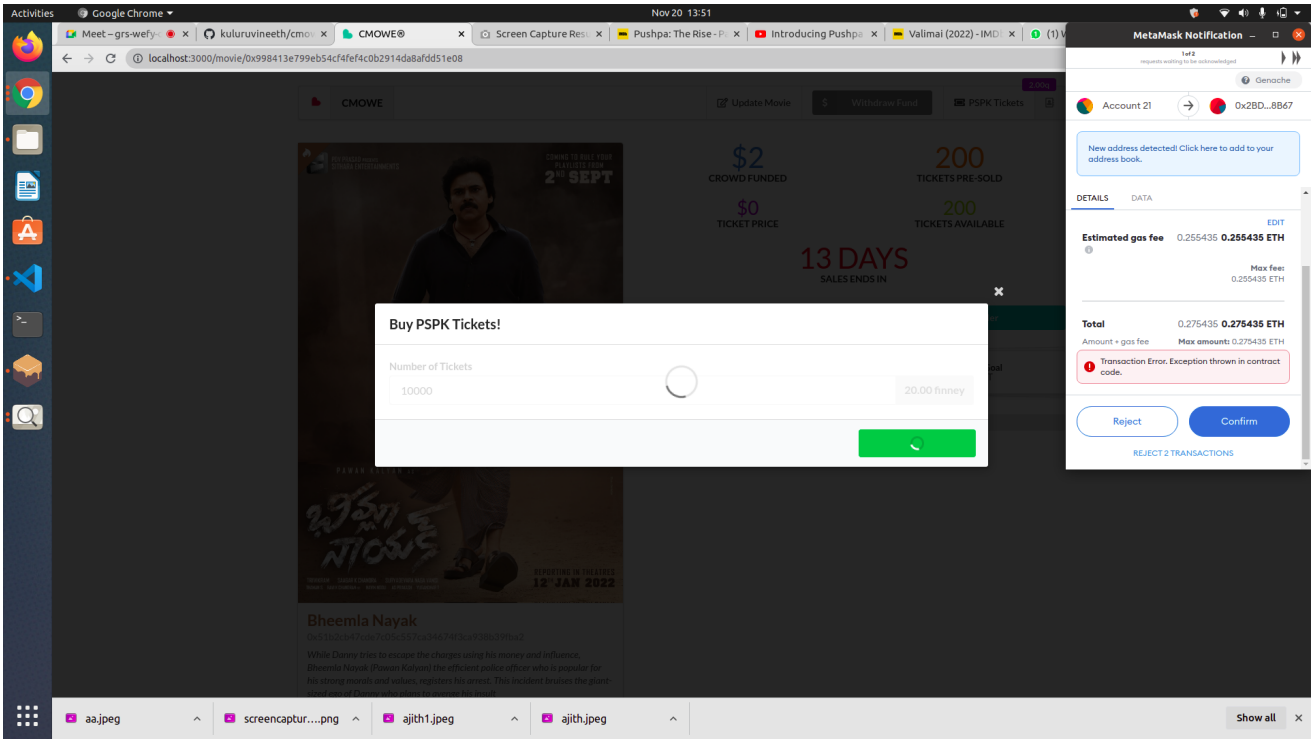
The movie page is unique to each project. There are three sections organized by function for ease of use. The left section displays the details of a movie in the form of a card for visual appeal. The ***movie poster*** is most prominent, followed by the ***title,*** the ***address of the filmmaker***, the ***logline***, the ***address of the token contract***, and then the ***date of creation***.



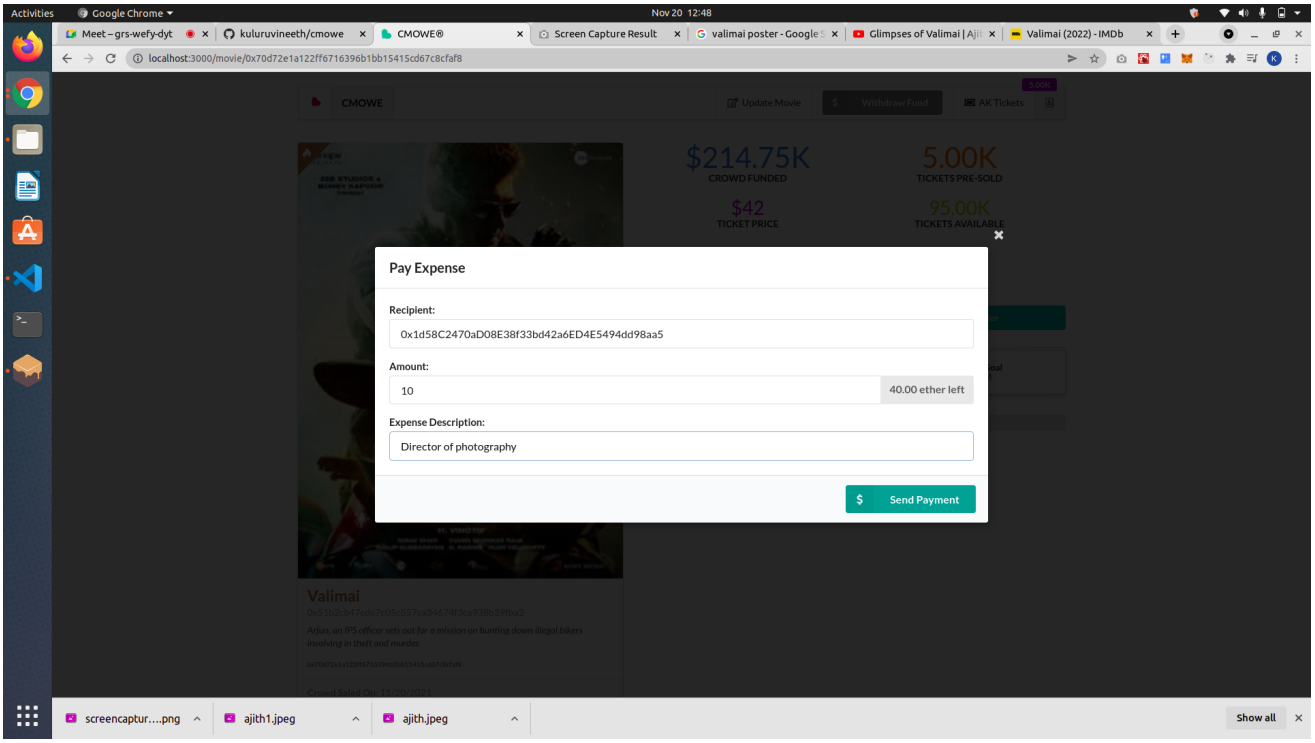
The right section highlights the most exciting **stats** about the project as well as information about the **token contract**. **Buttons** to purchase tickets and watch the trailer are prominently displayed. Underneath is a **progress bar** to indicate the percentage of tickets that have been pre-bought. Exact numbers can all be viewed on mouseover.



Clicking on **Buy Tickets** opens up a **modal** for the public to input the number of tickets they would like to purchase. For the buyer's convenience, the total cost in finney is automatically tallied on the right side. While the buyer confirms her intention with Metamask and the network processes the transaction, the form is disabled and a **spinner** is shown.



Again, if Metamask throws, the error message is passed to the buyer. The buyer's balance of tickets purchased is prominently displayed in the header. If too many, it will be shorthanded, but the exact quantity can be viewed on mouseover.

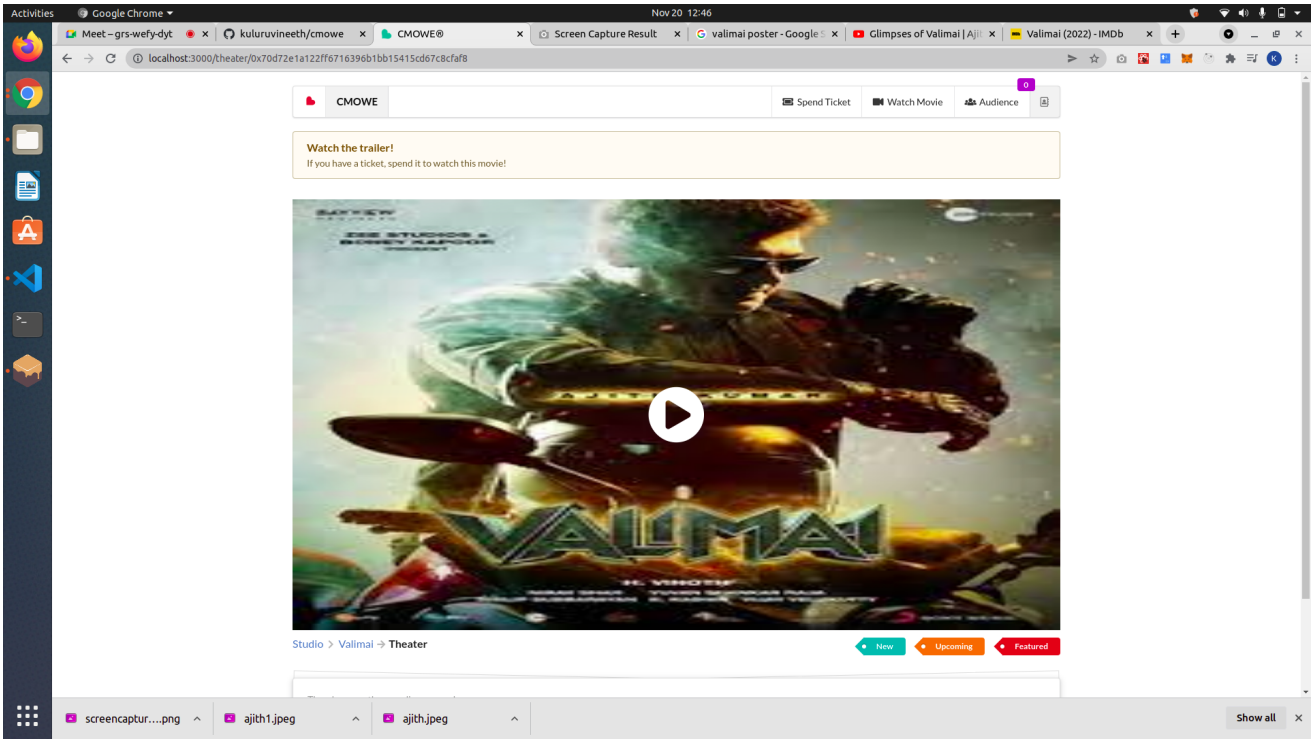


If the currently selected Metamask account is determined to be the filmmaker, a button to withdraw funds appears in the header. Clicking it opens this modal for the filmmaker to specify the recipient, amount, and description of the expense. For convenience, the remaining ether left after withdrawal is automatically calculated on the right side. While the filmmaker confirms her intent and the network processes her transaction, the form is disabled and a spinner is shown.

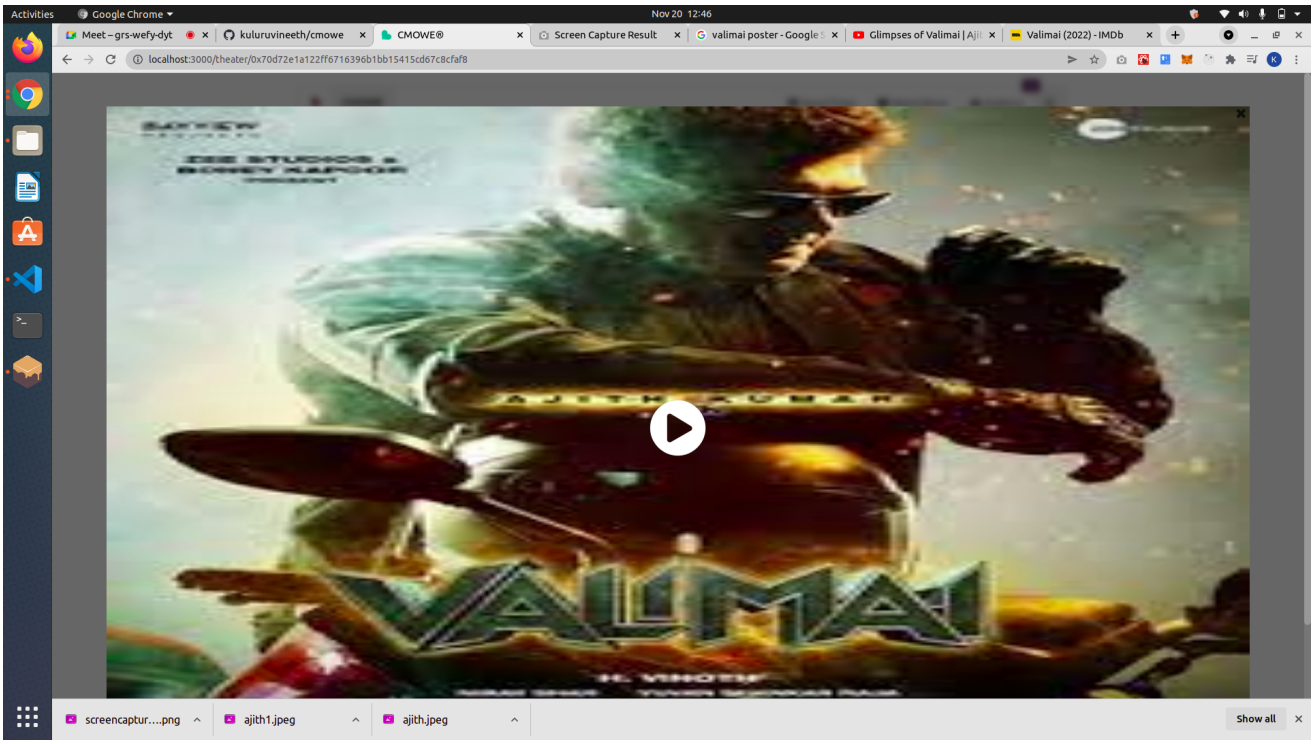
Finally, if Metamask throws, the error message is passed to the filmmaker to inform her of the reason.

6.4 Movie Theater Page

When the film is ready for commercial release, those with movie tickets can validate their ownership to access protected content. This page exemplifies how the blockchain securely verifies ticket possession.



When the public clicks on a **Watch Trailer**, they are taken to this page where they can watch the trailer to get excited about the movie. If the user is a ticket holder, there is also a **button** for her to spend her ticket. This action gives her access to watch the entire movie.For convenience, a **message** appears above the trailer to indicate if the user has permission to watch the movie or not. To elicit excitement, the total number of other people watching is shown in the header. Both the shorthand and the exact can be viewed. Below the trailer are the actual addresses of the people currently watching! .

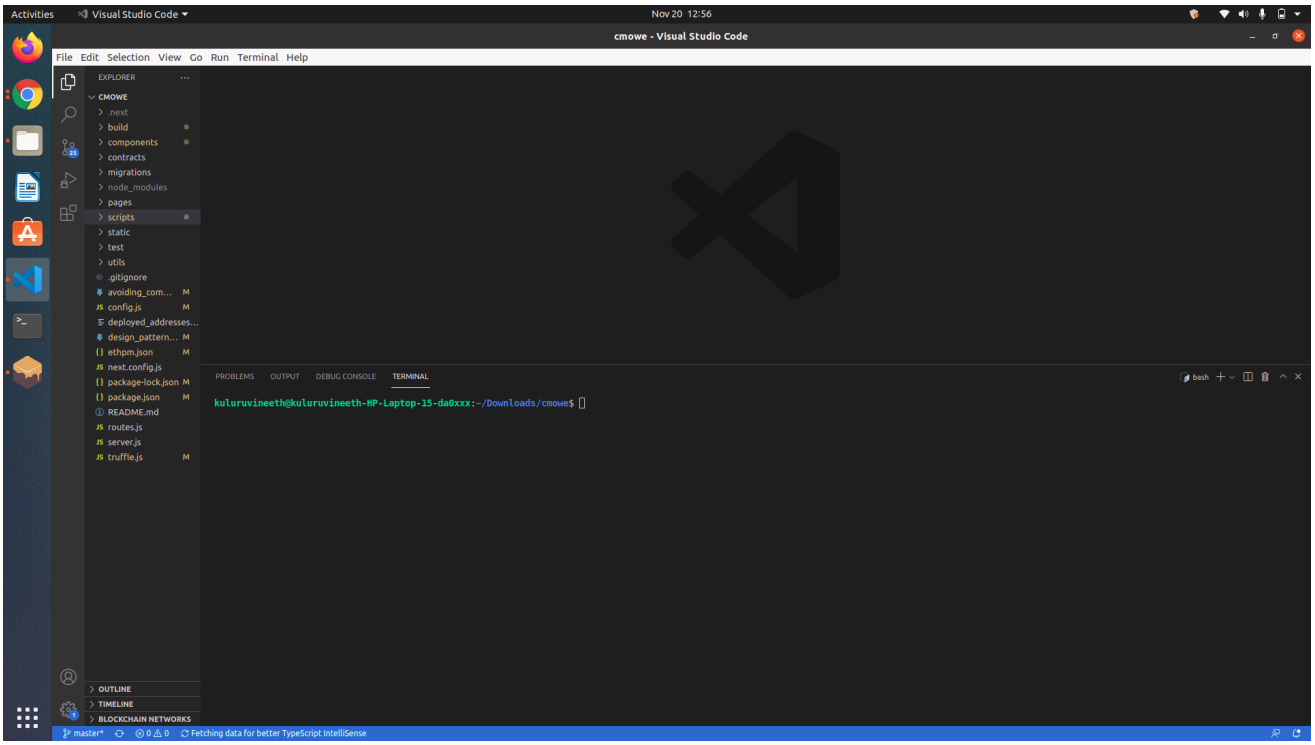


With the appropriate permission, a **Watch Movie** button appears. Clicking it dims the light and takes the viewer into theater mode!

7. Building the Architecture

Three architectural layers make up this dApp. On the backend, movie details are persisted on Ethereum instead of traditional databases. And to save on storage gas costs, movie posters are uploaded to IPFS via an Infura endpoint; only the hash that's returned is stored on the blockchain. To save on computational gas costs, secondary operations such as unit conversions, statistical calculations, logical checks, and transaction reorderings and filterings are offloaded to the client and the server as much as possible.

Another advantage of an off-chain server is the ability to fetch contract data and perform server-side rendering of the UI before page load. Because Ethereum is still nascent, in the likely event that Metamask or a web3-compatible browser is not installed, the site still shows pertinent data instead of blank fields.

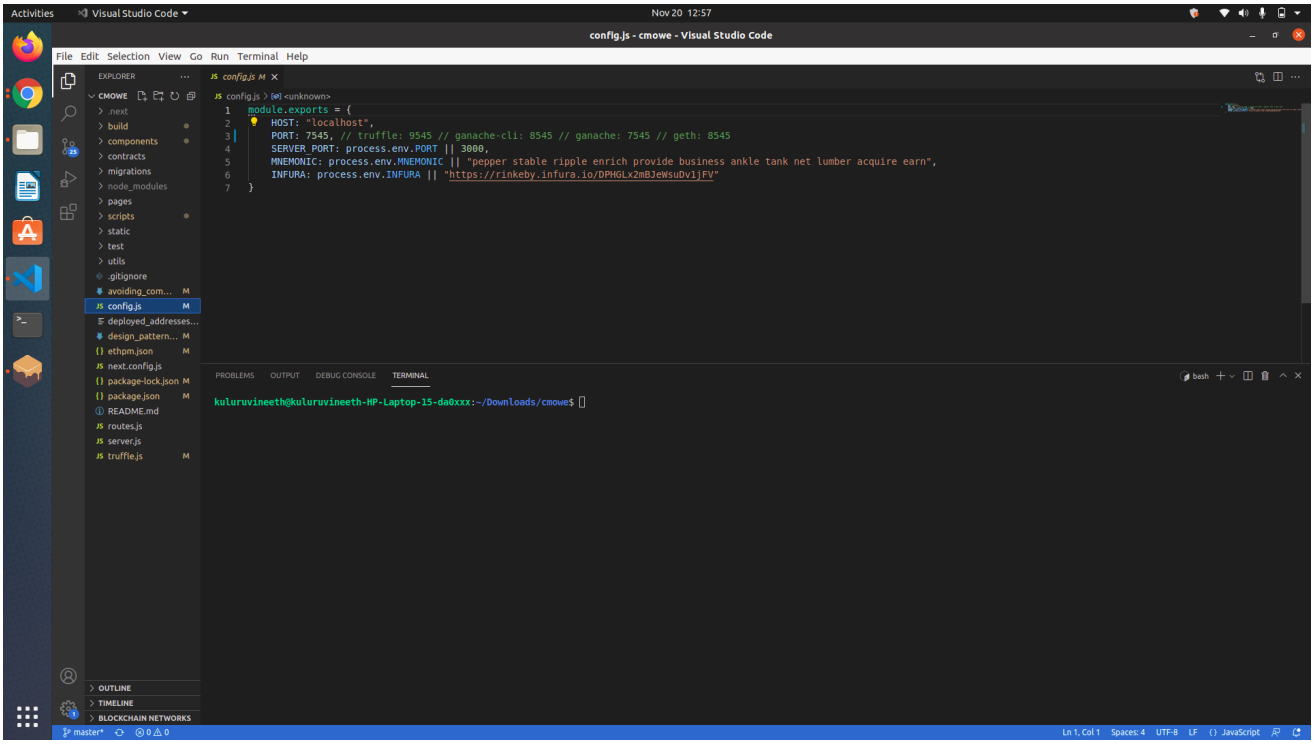


The way we organize our directories and files reflects our architecture. On the backend, the ***/contracts*** folder contains our solidity contracts. Inside ***/migrations*** are our compilation and deployment scripts, after which the bytecodes and ABIs are outputted to the ***/build*** folder according to the configuration in ***/truffle.js***. Truffle contracts that test our deployed contracts are found in the ***/test*** folder, as well as Truffle scripts that test our compiled contracts via ***web3***.

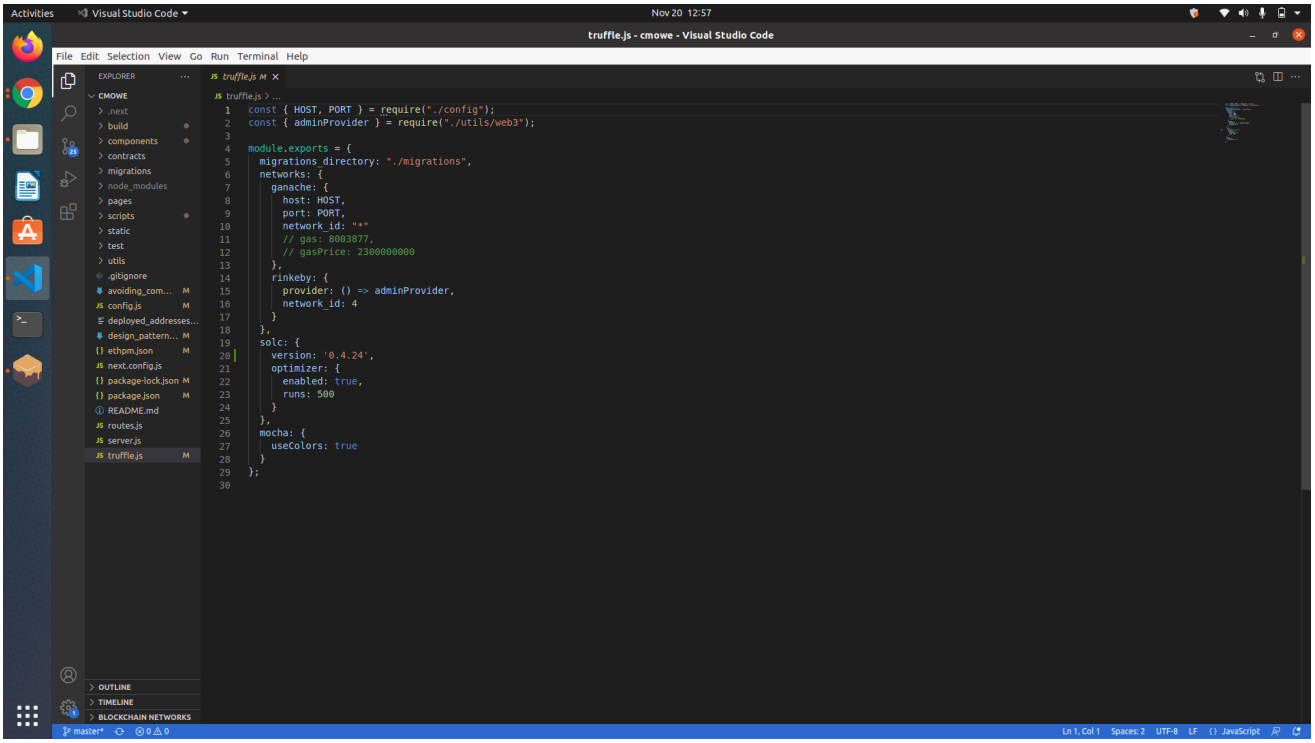
On the frontend, the four main pages are found inside ***/pages*** whose URL routes are specified in ***/routes.js***. Child components from which the four pages are composed are contained in ***/components***. Utility functions are extracted into modules inside ***/utils*** for reusability. Npm scripts that automate redundant processes are found inside ***/scripts***, while static images are located in ***/static***. Bundled resources are outputted to ***/.next*** according to the configuration in ***/next.config.js***. Configurations that affect both the backend contracts and the frontend server are encapsulated in /config.js for maintainability. Lastly, the server is launched according to /server.js.

7.1 Smart Contract Architecture

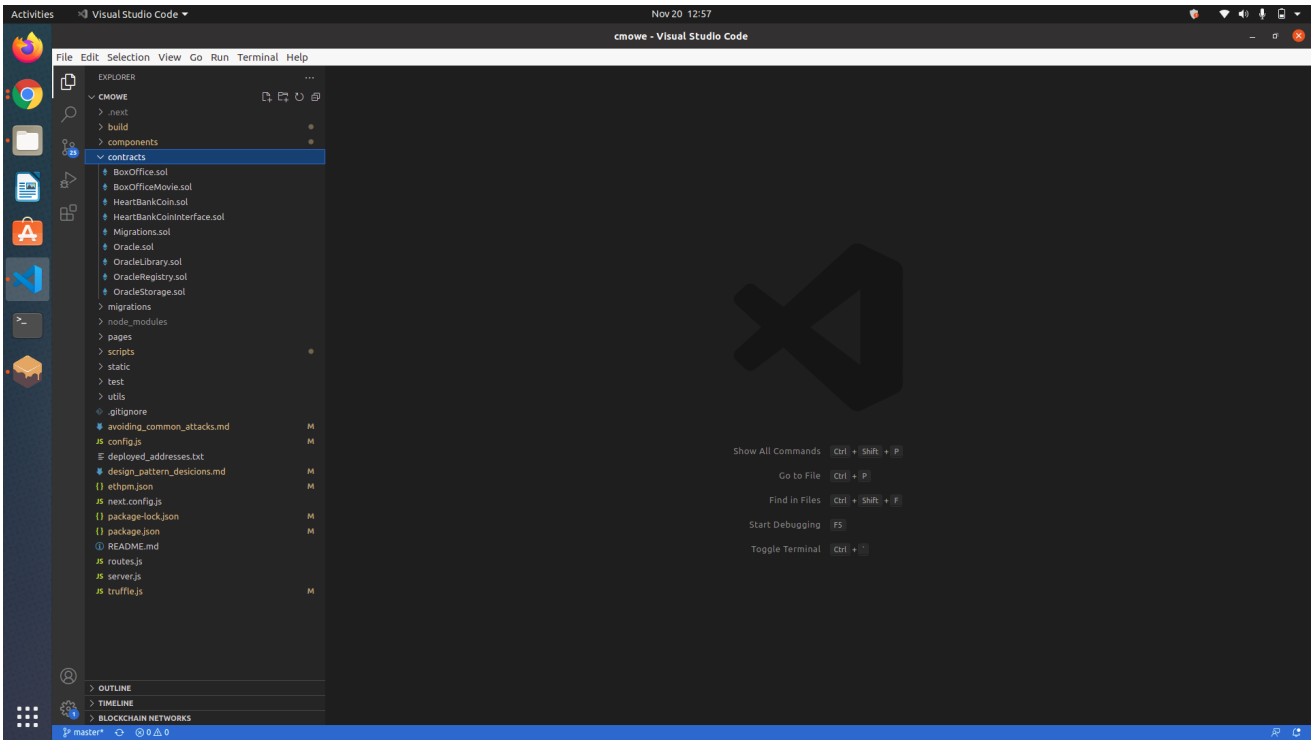
Three independent contracts underlie all ***web3*** calls inside our dApp. ***HeartBankCoin.sol*** is responsible for issuing Kiitos
coins. ***BoxOffice.sol*** is a factory tasked with creating ERC20 contracts per movie project, which ***BoxOfficeMovie.sol*** encapsulates and inherits from the ERC20 standard contract from OpenZeppelin. Finally, ***Oracle.sol*** communicates with an external API from Coinbase to update its internal ***price*** of ether for unit conversion to USD.



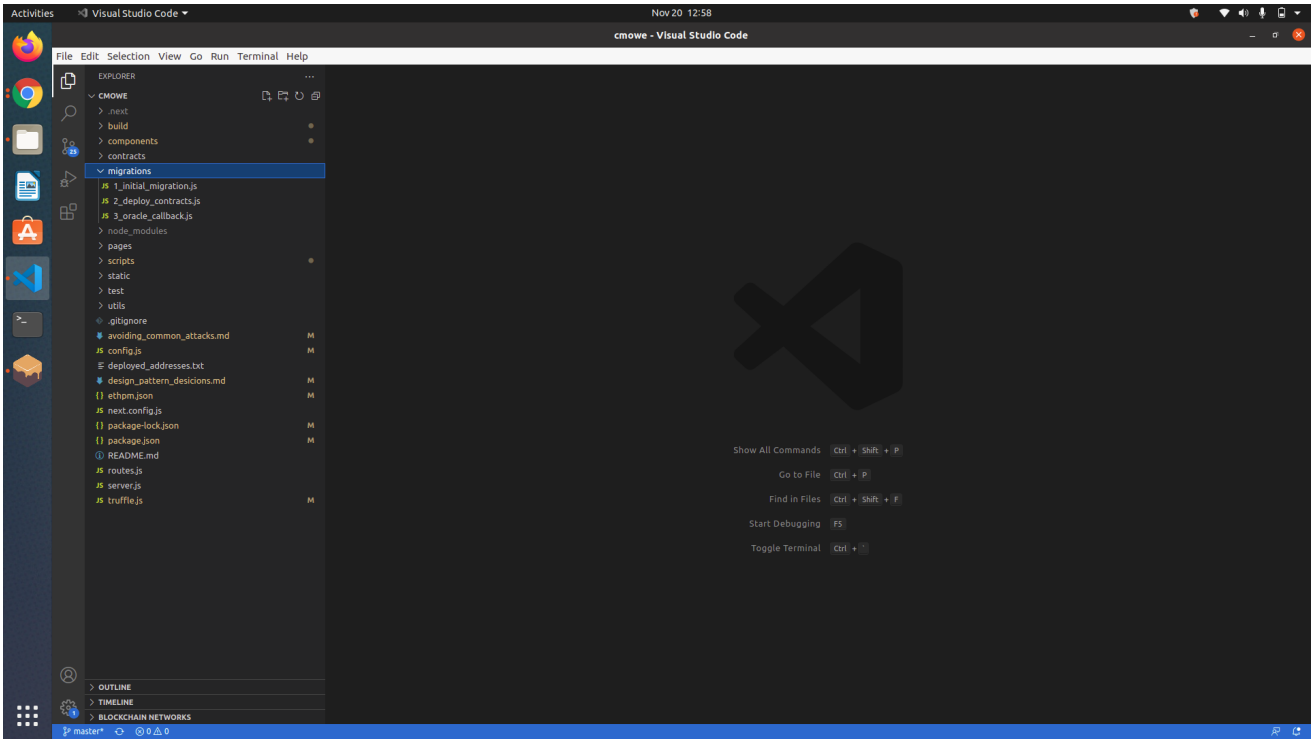
To make it easy to switch between local development networks and remote networks, the ***HOST, PORT, MNEMONIC, and INFURA*** provider are encapsulated in ***/config.js.***



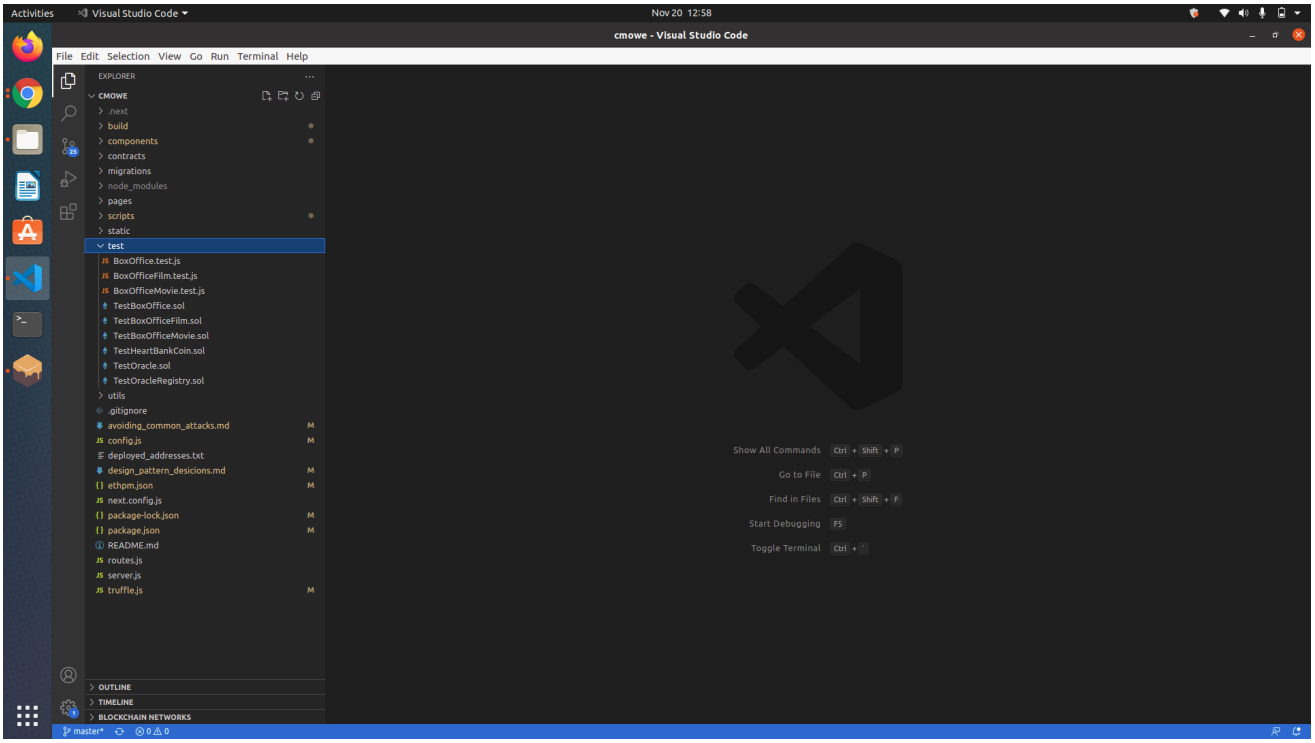
To make it easy to deploy to the Rinkeby Testnet, an additional network setting is added to the Truffle configuration file as ***rinkeby.***



BoxOffice.sol is a factory that instantiates ***BoxOfficeMovie.sol*** instances. ***HeartBankCoin.sol*** issues Kiitos coins that need to be purchased by filmmakers. ***Oracle.sol*** enables the admin to issue events that our oracle server can watch for before communicating with Coinbase to retrieve the current price of Ether in USD. It's architected according to the Upgradable Registry pattern whereby its logic and storage are decoupled from each other as ***OracleLibrary.sol*** and ***OracleStorage.sol***, respectively. The effect, ***Oracle.sol*** becomes upgradable, whose versions are tracked by ***OracleRegistry.sol***. Unrelated to our dApp, ***Migrations.sol*** is used by Truffle internally to manage the migration process.



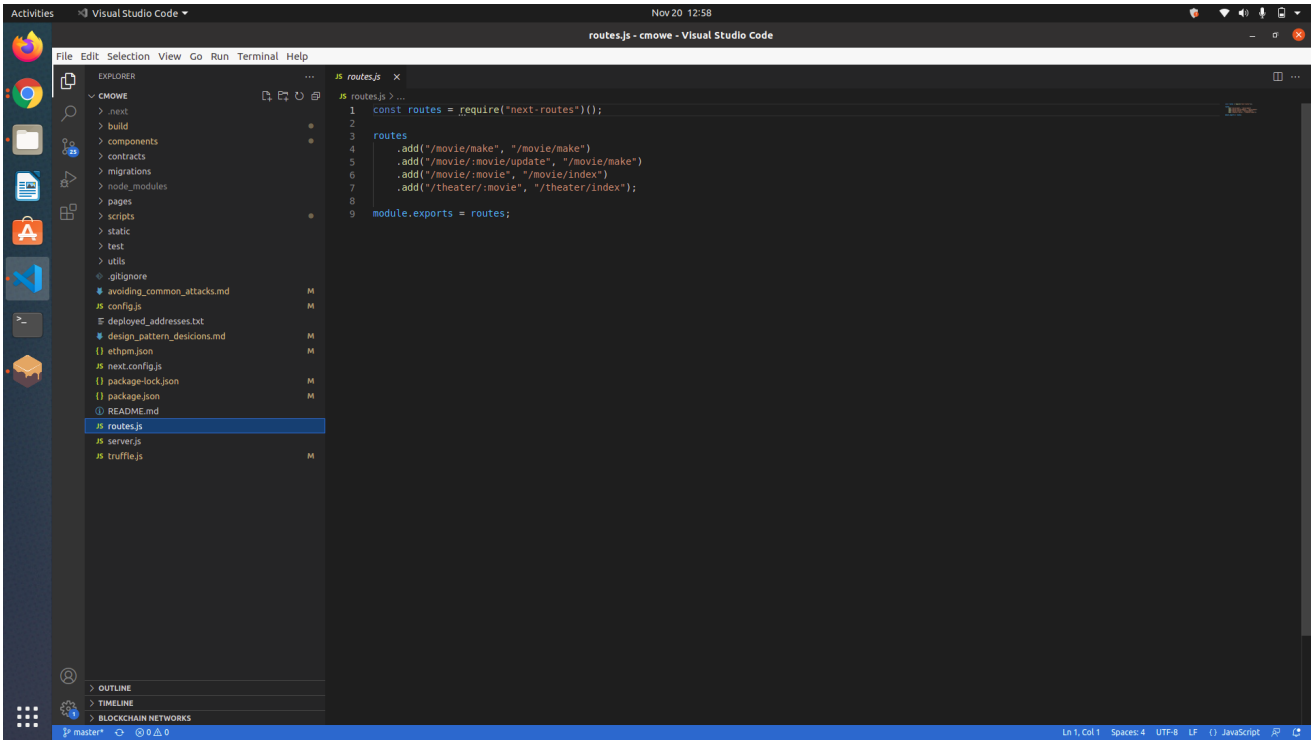
The first migration script is used by Truffle and is not related to our dApp. The second migration script deploys all our contracts, linking the necessary library contract and adding the necessary admin privileges after deployment success. The third migration script initializes the Oracle contract's USD price of ether by making a call to Coinbase's API and passing its value back to the contract.



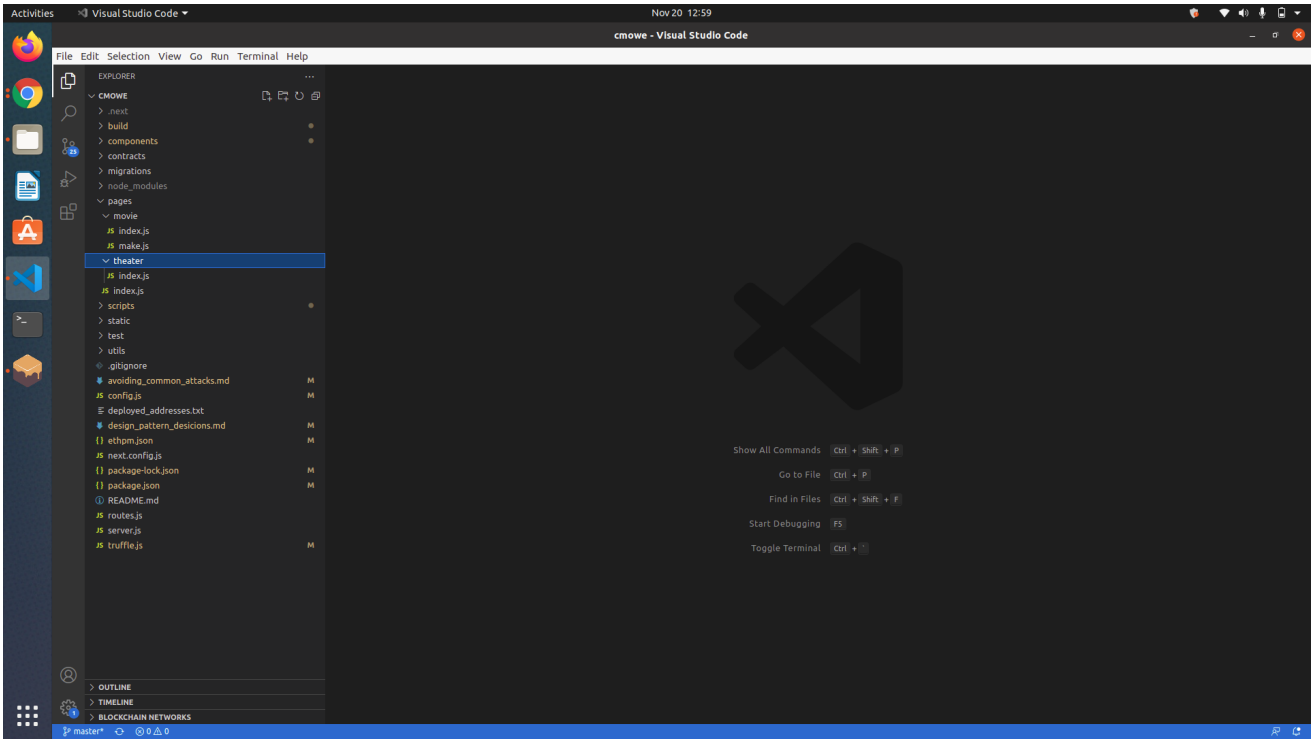
Truffle utilizes the Mocha framework to enable testing of contracts in both Solidity and JavaScript. To test each of our Solidity contracts, we create corresponding test contracts. Likewise, to test our Solidity contracts via web3 calls, we create corresponding JavaScript test files.

7.2 Frontend Architecture

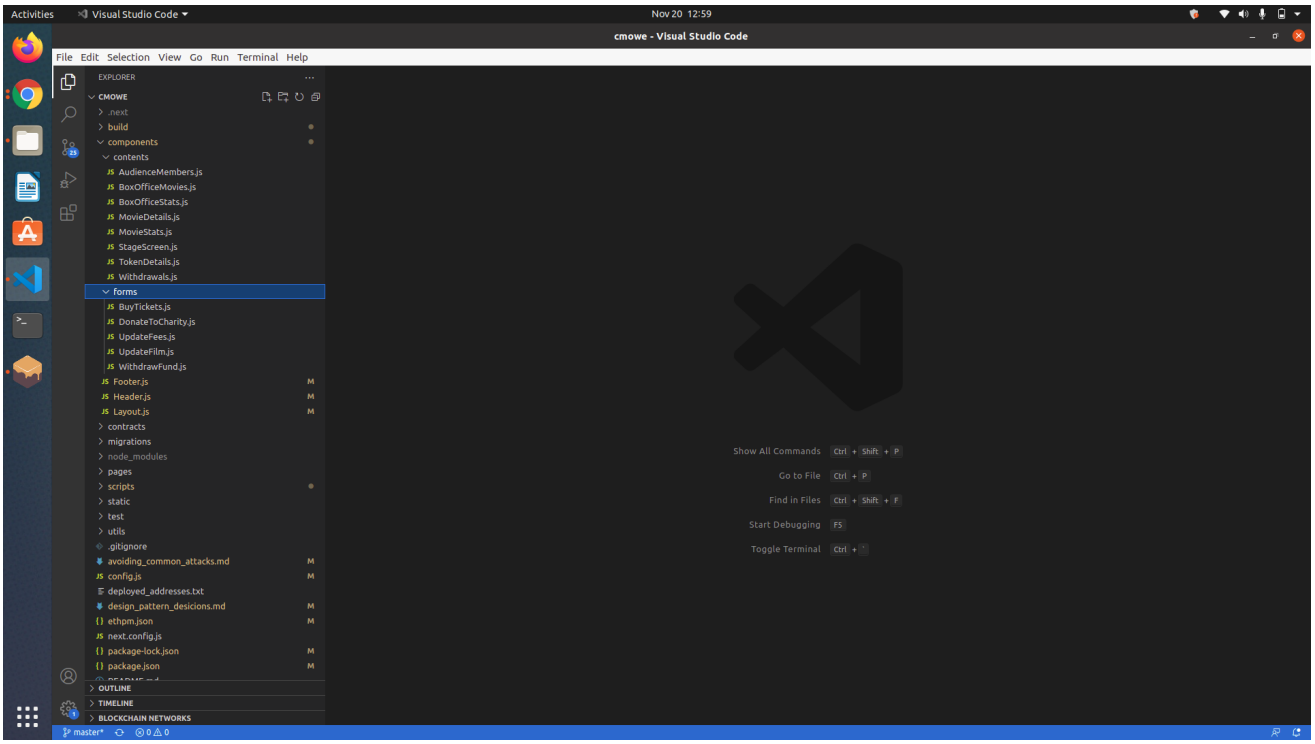
Four pages make up the frontend. Each is mapped to a **unique URL pattern** and **a React class component** that's capable of server-side rendering. The data it fetches on the server is passed down as ***props*** to the client and any child components. As such, the parent's ***state*** behaves like a Redux store, thus utilizing the Redux library would be redundant. In fact, with the release of the Context API in React 16, contract data can be passed in as a "store" on the parent level which makes it consumable at any level of the parent tree (i.e., children), much like a Redux ***Provider***.



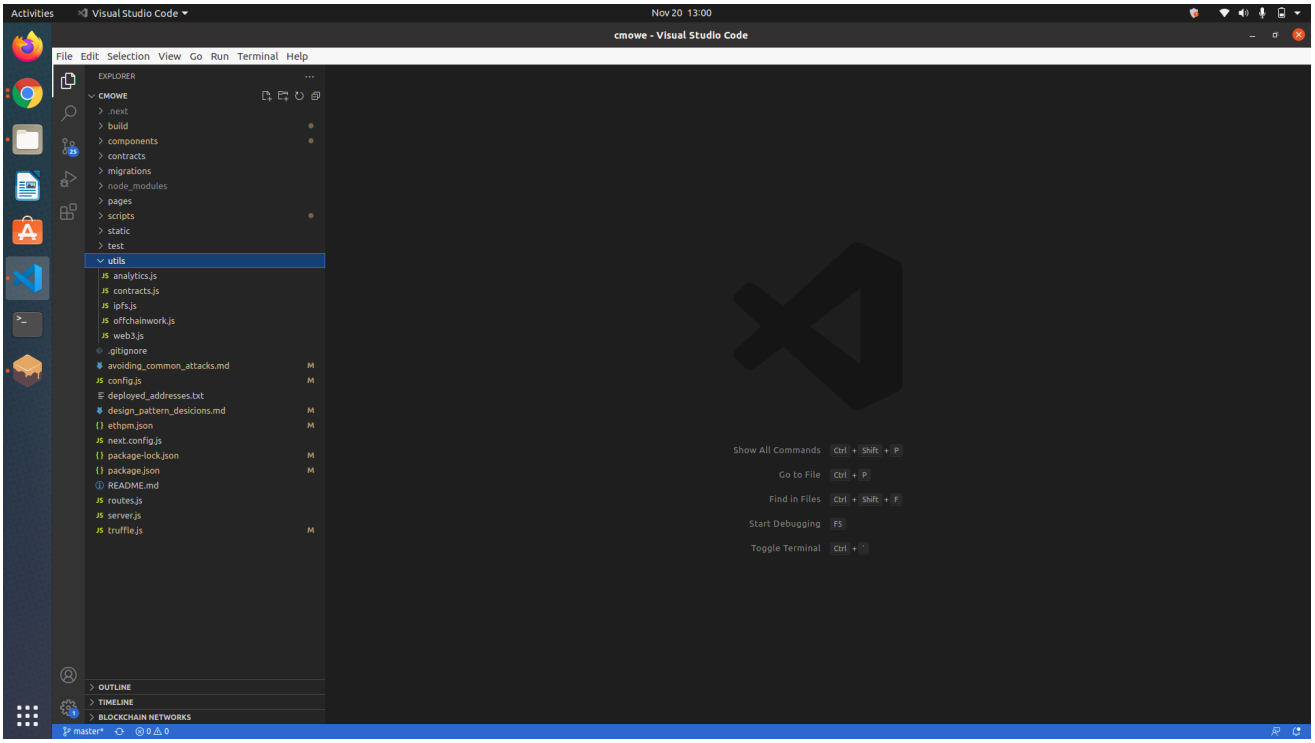
There are four main pages. Which routes map to which pages are specified in ***/routes.js***. The **studio home** page is simply at */*, but the **movie token** page, the **make movie** page, and the **update movie** page all share the path ***/movie/***. Because of their similar functionality, the make movie route and the update movie route share the same React class component. For bookmarking and search engine optimization, the movie token route and the update movie route are unique to each project, and therefore contain the ***address*** of the project. Similarly, the **movie theater** page also contains this dynamic parameter in its route.



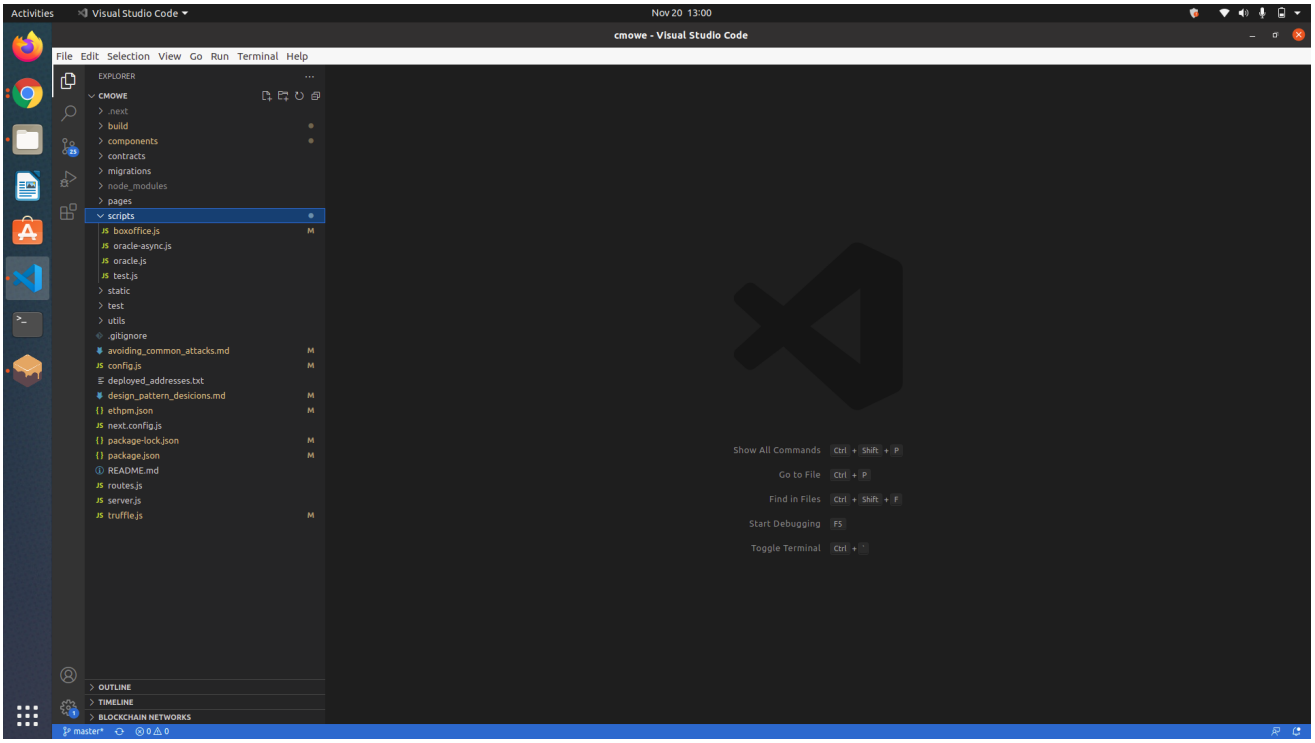
The actual React class components that the routes map to are found in ***/pages***. For symmetry, the folder and file structure exactly parallels the URL path.



The four main React components are composed with child components in a tree structure. They are found in ***/components*** and there are three types. Components whose function is to display data are functional components and are contained in ***./contents***. Components that submit data to the blockchain are class components and are contained in ***./forms***. The ***Layout, Header, and Footer*** are at the top-most level because they appear in every page. Each component is a tree structure in and of itself.



The communication bridge between the contract data and the frontend is located in ***/utils***. In ***./web3.js***, an instance of Web3 is instantiated with the necessary Provider. In ***./contracts.js***, the truffle-contract library is used to abstract the Truffle contracts that were compiled, together with the web3 instance to expose a user-friendly API surface. The connection to IPFS via an Infura endpoint is found in ***./ipfs.js***, and the communication with an oracle service is found in ***./offchainwork.js***.



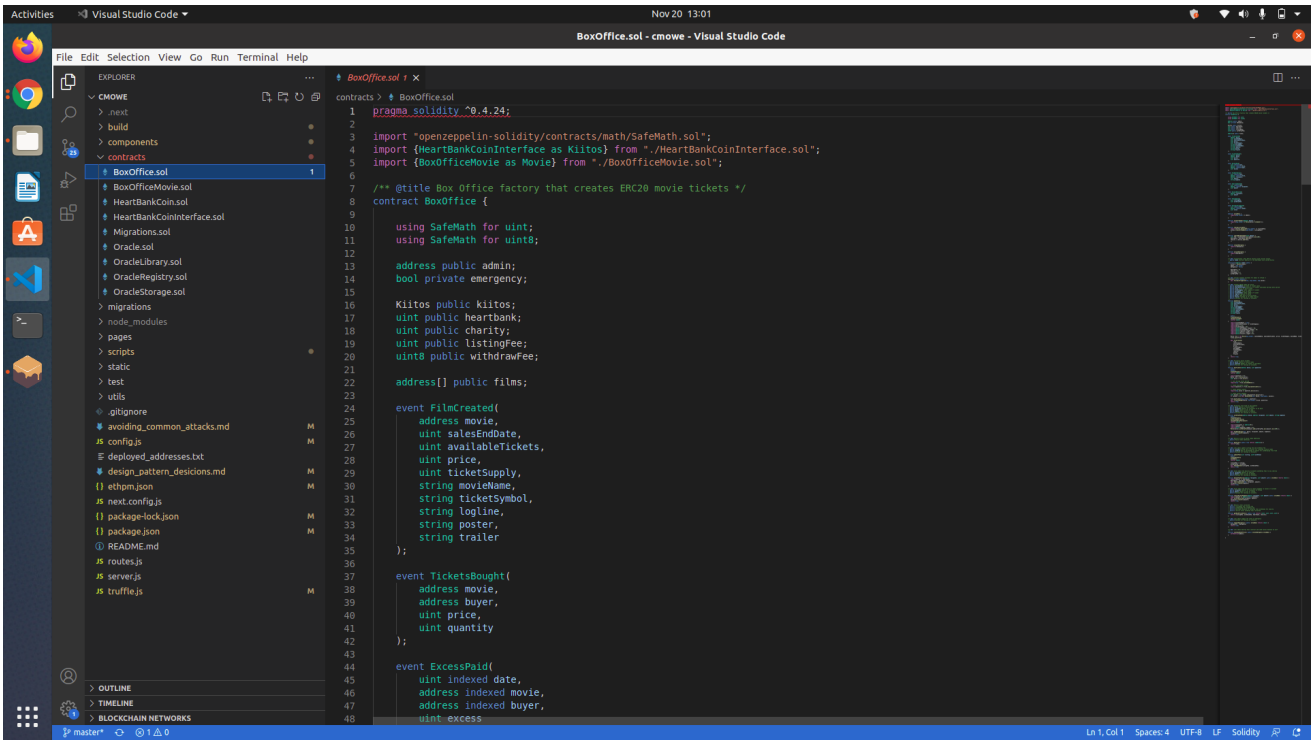
Helpful npm scripts are extracted to ***/scripts*** for ease of maintenance. The ***./oracle.js*** script can be used to mimic the functionality of an oracle. Executing it makes a call to the Oracle contract to emit an event that updates its ***price*** of ether in USD to the one currently at Coinbase. Due to conflict of

interest, we would delegate such calls to a third-party oracle service like Oraclize in a production setting if the stakes were high. Another useful script is ***./boxoffice.js***, which populates the **BoxOffice** contract with dummy data. Under the hood, these scripts take advantage of Truffle's ***exec*** command.

7.3 Smart Contract Design

1. BoxOffice Contract

The main ***BoxOffice.sol*** contract utilizes the factory pattern, from which ***BoxOfficeMovie.sol*** instances are created. Together, these contracts make up the core of our dApp. The others are ancillary. Instead of using a factory, we could have allowed our users to instantiate ***BoxOfficeMovie*** themselves, and just send in the address. This would save us from having to deploy ***BoxOffice***, but the users could easily tamper with the original code before instantiating. On the other hand, we could instantiate ***BoxOfficeMovie*** one-by-one ourselves on the backend, but this would mean we pay for the deployment cost each time. The factory pattern is an ideal middle because the original code is untouched and the users pay for the deployment which discourages abuse. Our only cost is the deployment of the factory.



We begin by importing the ***SafeMath*** library from OpenZeppelin to perform arithmetic operations that check for overflows and underflows. We also import ***HeartBankCoin's*** interface contract so that we can make calls to the ***HeartBankCoin's*** address by invoking its methods instead of making low-level calls to ***<address>.call()***. Just the interface suffices; there's no need to import the entire contract. Finally, we import ***BoxOfficeMovie*** for our factory to instantiate.

To follow convention, we start our contract by declaring the state variables it uses. For ***uint*** and ***uint8***, we make SafeMath's methods available with the ***using*** keyword. We store admin's address in ***admin*** and give her the ability to toggle the ***emergency*** state of the contract. For now, one admin suffices. Later, we may consider a multi-sig pattern with multiple owners for increased security.

We store HeartBankCoin's address in ***kiitos***, keep track of fees collected for charity in ***charity***, and its remaining balance in ***heartbank***. The number of Kiitos needed to create a film project is stored in ***listingFee***, and the percentage fee to withdraw funds from ticket sales is stored in ***withdrawFee***. The address of each BoxOfficeMovie instance created is appended to the ***films*** array.

For each function that updates the ***state***, a corresponding ***event*** is emitted so that we have the option to create a ***watcher*** for it if ever the need arises. Those that are likely to be retrieved for posterity and a cheap form of storage are ***indexed*** to enable faster queries. If gas costs are shown to outweigh the benefits, we may eliminate some events

Logics that are likely to be reused are extracted as ***modifiers***. Logics that are critical are also extracted to use as ***decorators*** for clarity of intent.

HeartBankCoin's ***address*** is passed in to our ***constructor()***, which initializes all our state ***variables*** upon creation. Ether sent to our contract in error is captured by our ***fallback()***, which emits the ***FallbackTriggered*** event to let the admin know who and how much to reimburse.

The ***makeFilm()*** function is our factory, which takes in the details of a film project and passes hem to the ***BoxOfficeMovie*** contract for instantiation. User inputs are validated for correctness beforehand. When the ***address*** of the newly created instance is returned, it's pushed into the ***films*** array and the ***FilmCreated*** event is fired.

The ***buyTickets()*** function enables the public to purchase as many movie tickets as they desire. Before the transaction is processed, however, the payment amount is checked as well as the ticket availability and sales period. Payment in excess is recorded for reimbursement should the buyer complain.The ***withdrawFund()*** function enables filmmakers to pay off the expenses of a film project by using the income from ticket sales. This process is transparent so that ticket holders can verify the recipients if they so choose, which gives credibility to a project and increases its value.

The function **getFilms()** function returns the entire ***array*** of film project addresses, which is needed by the frontend to display all the active projects. Though an automatic getter function is created for the ***film's*** variable, it expects an ***index*** as a parameter, returning only one address at a time.

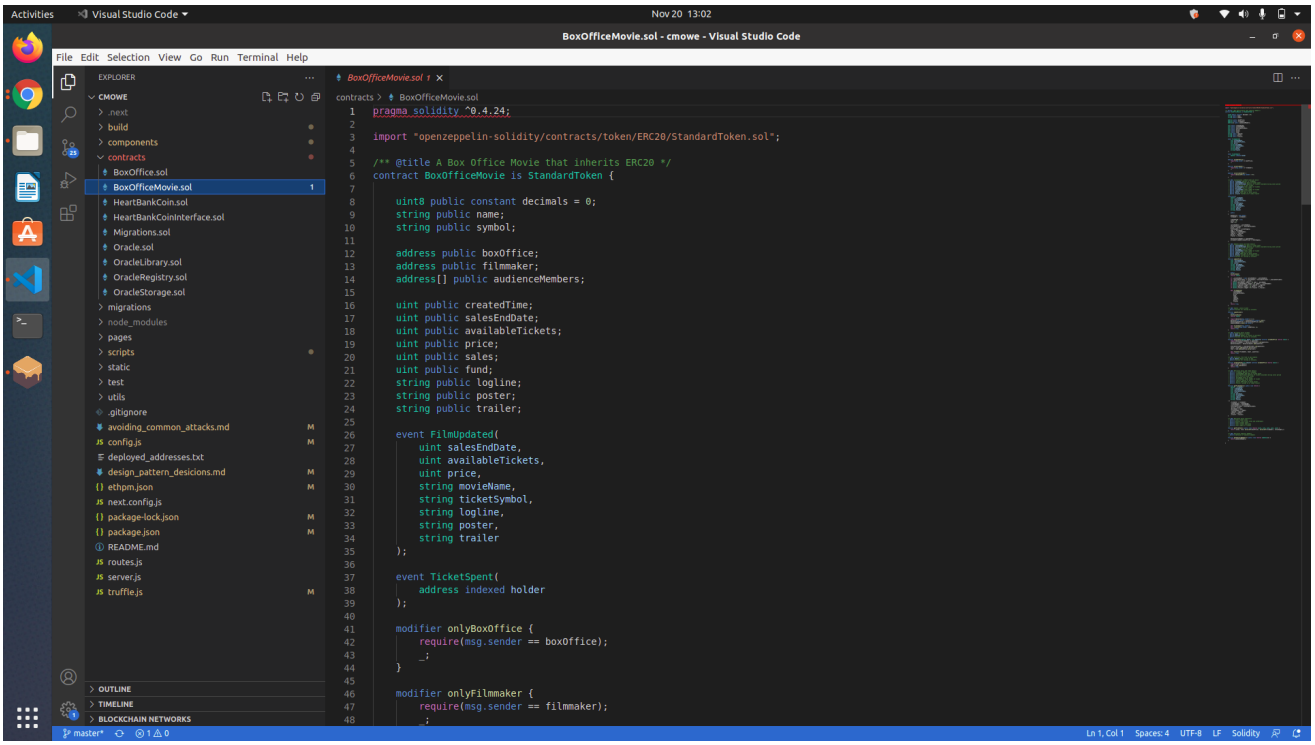
The function **updateFees(uint listing, uint8 withdraw)** function is used by the ***admin*** to update the listing fee and Withdraw fee. The function **donateToCharity(address recipient, uint amount)** function is also used by the ***admin***. With this function, the admin can donate the withdrawal fees collected to any

charity ***address***. The ***admin*** can use this function to return payments in excess should the buyer demands it. She can also use it to return erroneous payments captured by the ***fallback().***

Though automatic getter functions are created for these state variables, we want to make it possible for the frontend to retrieve these values with only one call for increased performance. With this function, the ***admin*** can toggle the ***emergency*** state of the contract.

Finally, with the function **shutDownBoxOffice()**, the ***admin*** can destroy the contract, sending all its ether to herself. This is quite dangerous and vests too much trust in the admin. After adequate user feedback, we may eliminate this functionality to better align with the spirit of decentralization.

2 BoxOfficeMovie Contract



We begin by importing the ERC20 standard contract from OpenZeppelin, which has been internally audited and externally vetted by the Ethereum community.

For convention, we start our contract by declaring our state variables, mindful that the data structures we choose do not force us to iterate through an array of unknown size. Though optional, we adapt ***decimals***, ***name***, and ***symbols*** to represent properties of a film project. Since it doesn't make sense to have fractions of movie tickets, ***decimals*** are hardcoded to ***0***. ***name*** is adapted to represent the title of a movie, and ***symbol*** can be any acronym the filmmaker desires for her movie tickets.

The calling BoxOffice's address is stored in ***boxOffice***, the filmmaker's address is stored in ***filmmaker***, and the audience members' addresses are stored in ***audienceMembers***, for permission checking. An audience member is someone who has both bought and spent a movie ticket.

salesEndDate is when a sales period ends, ***availableTickets*** is the number of available tickets, and ***price*** is the cost in wei of each one. There can be as many sales periods as the filmmaker desires. She just needs to update the ***salesEndDate, availableTickets, and price***, accordingly. Ticket sales are recorded in ***sales***, and the remaining balance is accounted for in ***funds.*** Most likely, the sales periods will correspond to the various stages of film production.

createdTime is when the project was created, and ***logline*** can be as long as the filmmaker likes which she pays for in gas. ***poster*** is the IPFS hash of the movie poster image, and ***trailer*** is the YouTube ID of the movie trailer video. (Uploading video files to IPFS takes quite some time!) Later, other video sources may be considered. This contract has two functions that update the ***state***. Therefore, ***events*** are created for them in case there are ***callbacks*** watching. ***TicketSpent*** is ***indexed*** because it's likely to be searched through.

To make permission checks clear and obvious, modifiers are created for ***onlyBoxOffice, onlyFilmmaker, and onlyTicketHolder.*** The ***BoxOffice*** factory calls this ***constructor()***, passing along all its arguments, to create an instance. As the caller, ***msg.sender*** is now BoxOffice's address. The originating externally owned account is passed in as the ***filmmaker***. By convention, the filmmaker is given all the supply of tickets. For BoxOffice to be able to initiate transactions on the filmmaker's behalf, it's given the full allowance.

```
function updateFilm(
    uint _salesEndDate,
    uint _availableTickets,
    uint _price,
    string _movieName,
    string _ticketSymbol,
    string _logline,
    string _poster,
    string _trailer
 )
```

With this function, filmmakers can update any movie detail except the total supply of tickets. The only requirement is to pass along all other movie details. It's a simpler approach than having to create setters for each state variable. Based on user feedback, we may need to revisit this strategy. Regarding the permanence of total supply, enabling the minting of more tickets causes inflation, which decreases the value of each one, and the overall credibility of the project.

The function **buyTickets(address buyer, unit quantity)** is called from the ***BoxOffice*** factory in order to collect payments in one place for easier management. After ***BoxOffice*** performs the necessary checks, this function is called to perform the actual ticket transfer from ***filmmaker*** to ***buyer***. ***availableTickets, sales***, and ***funds*** are updated accordingly.

When ticket holders are ready to spend a ticket to gain access to the movie, they can call this function. Their ticket is then transferred to ***boxOffice***, which holds all the spent tickets.

The function **withdrawFund(uint amount)**is also accessed from the ***BoxOffice*** factory, where the payments are collected. When filmmakers make a withdrawal, this function is called by ***BoxOffice*** to update their ***fund*** balance.

Though automatic getters are created for these public variables, we want to provide a way for the frontend to make only one call for increased performance.

Because the EVM stack can only hold so much, we have to create this additional getter function to return other pertinent movie stats.

Finally, we return the entire ***audienceMembers*** array for the frontend to use in its validation logic to check which users have the necessary perFinally, we return the entire audienceMembers array for the frontend to use in its validation logic to check which users have the necessary permission to watch a movie.mission to watch a movie.

7.4 Solidity and JavaScript Unit Testing

There are two types of Solidity functions. Those that *update state*, and those that return state. For those that update state, a ***bool*** is returned to assist in asserting for success. To create the necessary permission to access ***onlyAdmin*** and ***onlyBoxOffice*** functions, the ***new*** keyword is used to create an instance of the necessary contracts.

1. TestBoxOffice.sol

Here we begin by setting ***initialBalance*** to ***1 ether*** to give our contract the necessary ether for testing, which is a feature of the Truffle testing framework.

testInitialState()

For our first unit test, we want to make sure that the ***constructor()*** function correctly initializes all the state variables. Unfortunately, ***withdrawFee*** is ***uint8***, which the ***Assert*** library does not support.

testFallBack()

Next, we want to test that our ***callback*** function is able to catch calls that do not match any function signatures in our contract. We achieve this using the low-level ***call*** method built-in to our contract ***address***.

testUpdateFees()

Here, we want the ***listingFee*** and the ***withdrawFee*** to update correctly. Only the ***admin*** can call this function, so we use the ***new*** keyword to instantiate a ***BoxOffice*** instance to give ***TestBoxOffice*** the necessary permission.

testReturnExcessPayment()

Here, we want to make sure that the ***admin*** can successfully refund extraneous or erroneous payments (captured by the fallback). Again, the ***new*** pattern is used to grant the test contract the appropriate permission.

testShutDownBoxOffice()

We also want to make sure that the ***admin*** can successfully destroy a ***BoxOffice*** instance. We have to toggle the ***emergency*** state to ***true*** before executing the call to ***self destruct***. Because the EVM cannot return a ***boolean*** to let us know that the self-destruction was successful, we resort to a low-level ***call*** to assert for success.

2. TestBoxOfficeFilm.sol

This test contract is an extension of ***TestBoxOffice***, extracted to ease maintenance because all test functions require a ***BoxOfficeMovie*** instance. As such, we take advantage of the ***beforeEach()*** helper method to call BoxOffice's ***makeFilm()*** to instantiate a ***BoxOfficeMovie*** instance before each unit test.

Again, we initialize our test contract with ample ether using initialBalance offered by the Truffle testing framework.

testBuyTickets()

For this unit test, we want to make sure that anyone with an Ethereum wallet address can purchase as many movie tickets as he or she desires within the limits of the sales period and availability.

testWithdrawFund()

Here, we want to make sure that filmmakers are able to send payments using the funds they've collected from ticket sales.

testDonateToCharity()

Finally, we want to ensure that the admin can successfully donate the fees collected to any recipient address.

3. TestBoxOfficeMovie.sol

Again, we take advantage of the ***beforeEach()*** helper to instantiate a ***BoxOfficeMovie*** instance before each test, using the ***new*** keyword to give the test contract the required permission.

testMovieDetails()

Firstly, we want to make sure that all movie details are correctly stored by the ***constructor()*** function, and that the automatic ***getter()*** functions actually work for all our ***public*** variables.

testUpdateFilm()

Here, we want to ensure that the movie details can be updated by the filmmakers.

testSpendTicket()

Here, we want to make sure that ticket holders are able to spend the tickets they have purchased, thus giving them the necessary permission to watch the desired movies. After which, these tickets should no longer be tradable.

testWithdrawFund()

Though only the ***BoxOffice*** contract can call this function, we test it nonetheless for completeness. Here, we want to make sure ***BoxOffice*** can call ***withdrawFund()*** to make the necessary ledger balance changes.

Because Solidity contracts cannot retrieve emitted events at runtime, they cannot be tested in Solidity. On the JavaScript side via ***web3***, however, this is not the case. Knowing this, we design our test functions to take advantage of these events, catching them to assert for correctness. Also, unlike Solidity tests, ***state*** is retained throughout testing. For completeness, JavaScript unit tests closely parallel those in Solidity.

4. BoxOffice.test.js

We begin by using the ***before()*** helper to retrieve the instance of ***BoxOffice*** that was deployed during Truffle migrations. We also initialize a ***Date*** variable for use, making sure to convert it from milliseconds in JavaScript to seconds since the unix epoch for the EVM.

it(“should store initial states”)

Like its Solidity counterpart, we want to make sure initial states are correctly stored and can be retrieved via ***web3***.

it(“should create film and movie tickets”)

Here, we want to make sure any filmmakers can use our contract to create a film project and issue movie tickets. To test for this, we create a ***callback*** function to watch for the ***FilmCreated*** event. Once caught, we declare our assertions.

it(“should update fees”)

To test that fees have been updated correctly, we create a ***callback*** to watch for the ***FeesUpdated*** event.

it("should return excess payment")

To test that excess payment has been returned correctly, we use web3's ***getBalance()*** method to check the balance of the recipient's account.

it("should receive plain ether transfer and trigger fallback")

To test that a plain ether transfer has successfully completed, we use web3's ***getBalance()*** to compare the sender's balance before and after the transfer. To check that the contract's ***fallback()*** is triggered, we create a ***callback*** to watch for the ***FallbackTriggered*** event.

it("should shut down box office and self-destruct")

To test that our contract instance has successfully self-destructed, we simply assert for an ***ok*** status because nothing can be returned after self-destruction. As long as no exceptions are thrown, the test passes.

5. TestBoxOfficeFilm.sol

Again, functions that require a ***BoxOfficeMovie*** instance are extracted to a separate module for maintainability. We begin by declaring a ***Date*** variable like before, and use the ***before()*** helper to create an instance of ***BoxOfficeMovie*** that we can use throughout our unit tests.

it("should purchase movie tickets with excess payment")

To test that movie tickets have been successfully bought and that payment was in excess, we create a ***callback*** to watch for the ***TicketsBought*** event and another ***callback*** for the ***ExcessPaid*** event.

it("should withdraw from fund")

To test that funds have been successfully withdrawn by the filmmaker, we create a ***callback*** to watch for the ***FundWithdrawn*** event.

it("should donate to charity")

To test that fees have been successfully donated by the ***admin***, we create a ***callback*** to watch for the ***CharityDonated*** event.

6. BoxOfficeMovie.test.js

Like before, we begin by declaring a ***Date*** variable and creating an instance of ***BoxOfficeMovie*** that we can use throughout our tests with the help of ***before().***

it(“should get movie details”)

Like its Solidity counterpart, we want to make sure movie details are correctly stored and can be retrieved via ***web3***.

it(“should get film summary”)

We also want to make sure that all the movie details can be retrieved with just one call to ***getFilmSummary()*** to increase performance on the frontend.

it(“should update film”)

To test that movie details can be successfully updated, we create a ***callback*** to watch for the ***FilmUpdated*** event.

it(“should spend movie ticket”)

To test that a ticket has been successfully spent by a ticket holder, we create a ***callback*** to watch for the ***TicketSpent*** event.

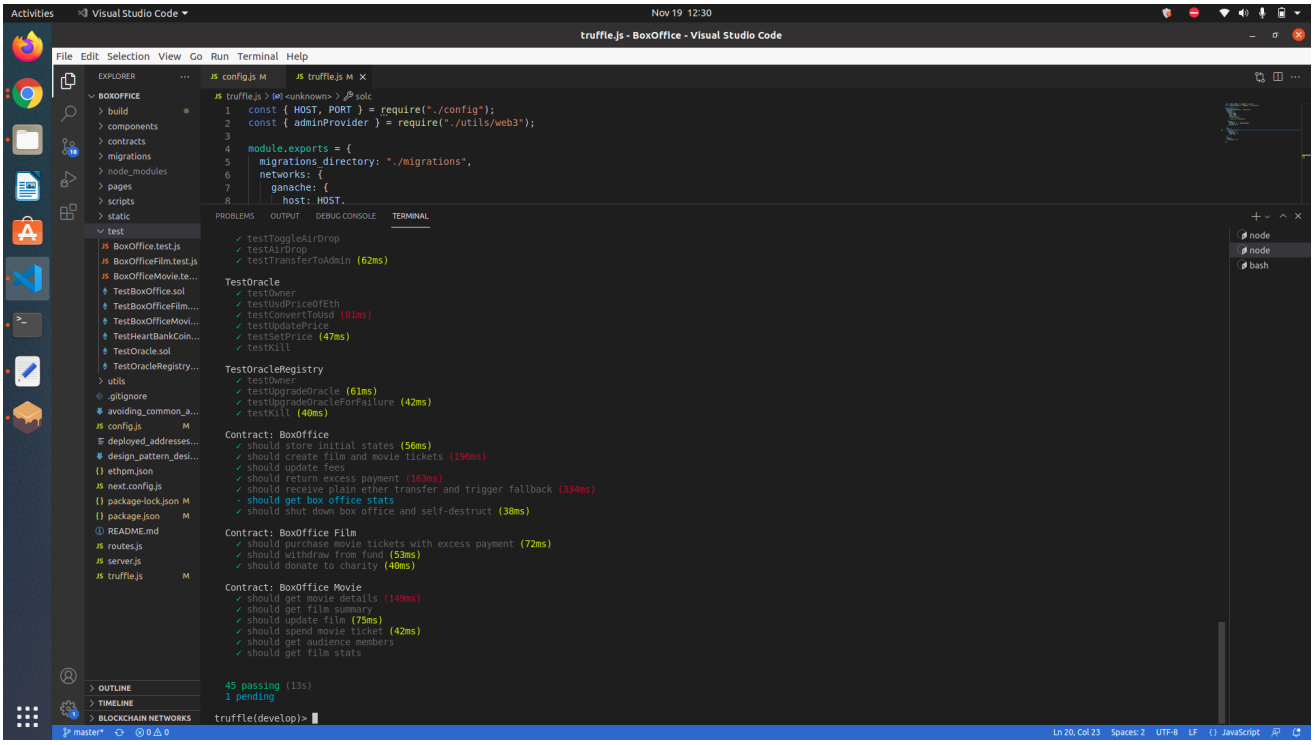
it(“should get audience members”)

Here, we want to make sure that once a ticket is spent, the ticket holder becomes an ***audience*** member, thus acquiring the necessary permission to watch the desired movie.

it(“should get film stats”)

With just a single call, we want the frontend to be able to retrieve useful statistics about any film project. Fortunately or unfortunately, the typical web surfer expects a snappy UI experience.

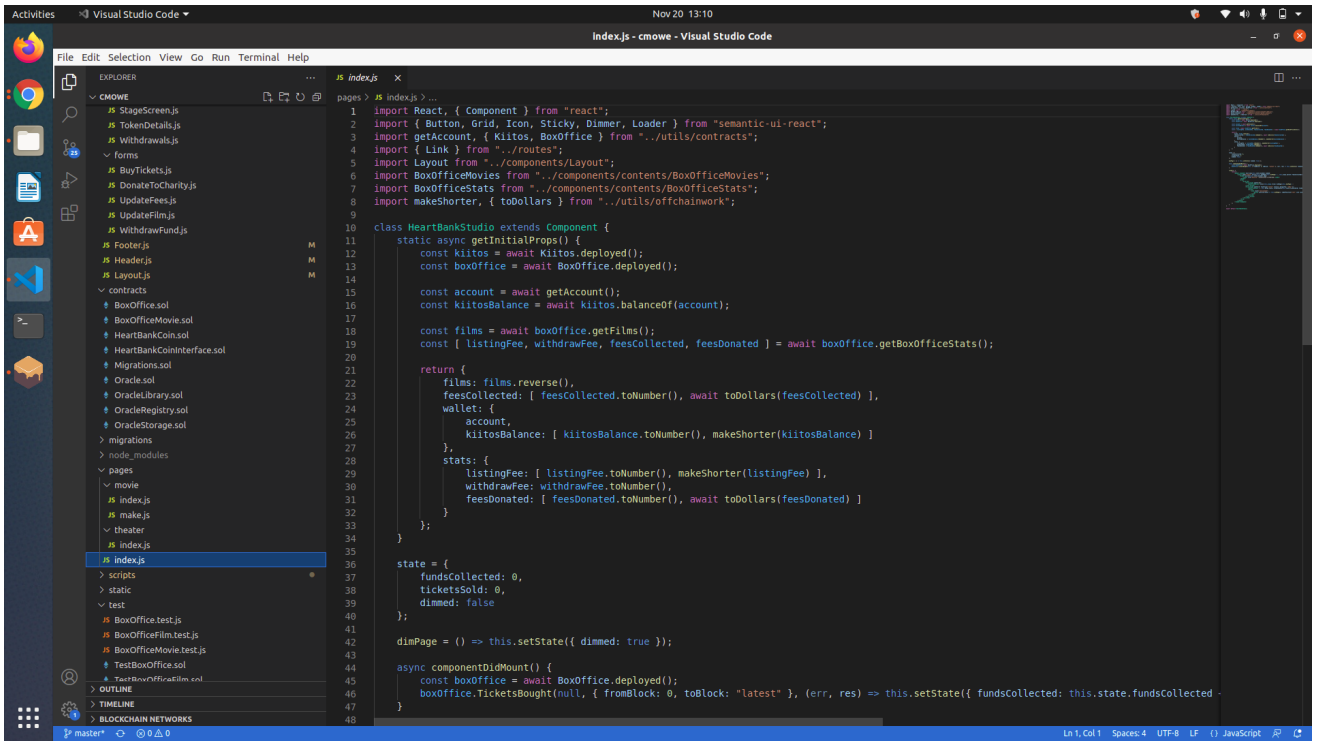
Unit Test Results



7.5 Frontend Design

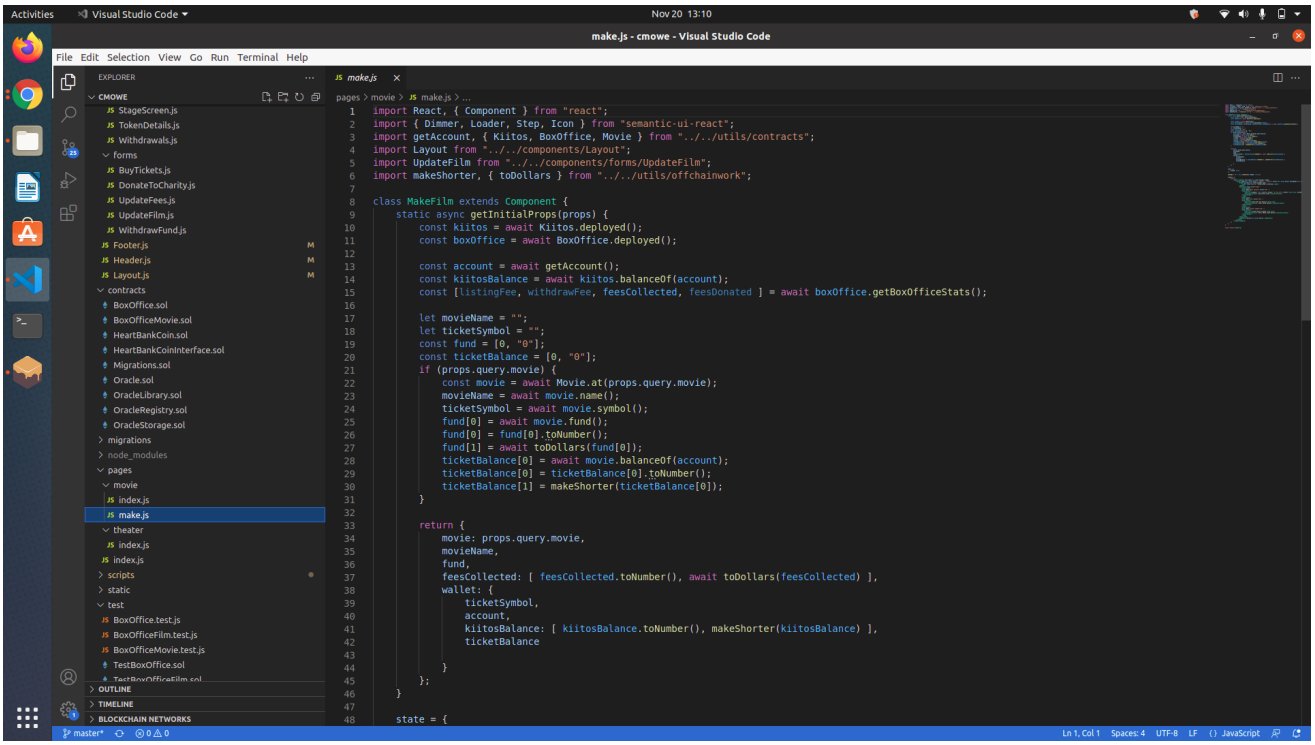
Composition is favored over inheritance. There are four pages and each is a **parent class component**, and there are two types of child components. Forms need to manage internal state, so they are also class components; otherwise, they are simply functional components to increase performance. Child components are composed inside the parent components, and contract data is fetched at the parent level via ***getInitialProps()*** and passed down as ***props*** to the children.

1. HeartBankStudio Component



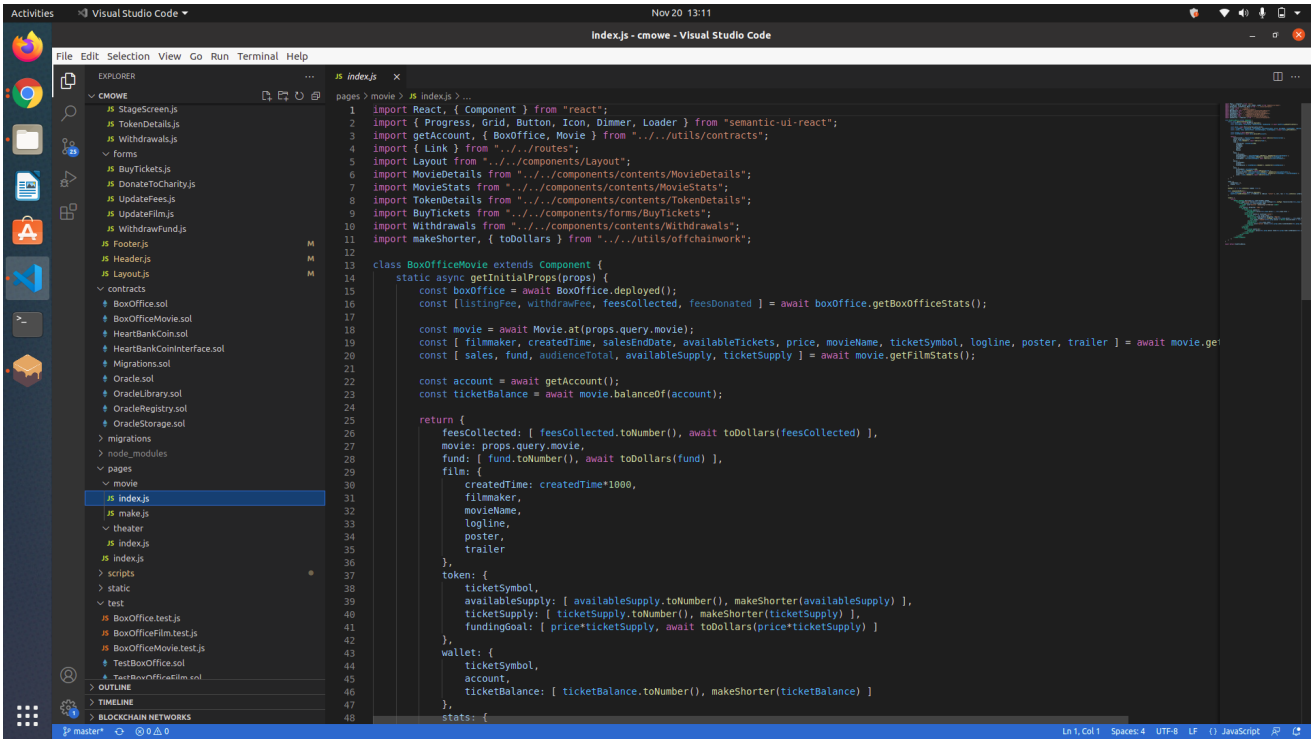
All contract data relevant to this page are retrieved in ***getInitialProps()*** which occurs server-side. First, we call ***getAccount*** to retrieve the user's currently selected Metamask ***account***. Then, we pass it along to ***kiitos.balanceOf(account)*** to get her Kiitos balance. We call ***boxOffice.getFilms()*** to get all the currently active film projects, and ***boxOffice.getBoxOfficeStats()*** to get all the exciting statistics. Finally, we create an object with all these values to return as ***props*** for the child components to consume, i.e., ***BoxOfficeMovies*** and ***BoxOfficeStats***.

2. MakeFilm Component



Again, all contract data retrieval occurs in ***getInitialProps()***. There, we try to retrieve the film project address from the ***URL path (props.query.movie)***. If it exists, we use it to make a call to ***Movie.at(props.query.movie)*** to render a page that lets the filmmaker update her film project. If it doesn't, we render a page that lets the filmmaker create a new project instead. The main child component, ***UpdateFilm***, also follows this logic to prefill the form with prior data if an address exists.

BoxOfficeMovie Component



Again, we begin inside ***getInitialProps()*** where we make a call to ***Movie.at(props.query.movie)*** to get the relevant instance of ***BoxOfficeMovie***. Having this instance, we can now call ***getFilmSummary()*** and ***getFilmStats()*** to retrieve all the pertinent facts of a film to pass along to ***MovieDetails***, ***MovieStats***, ***TokenDetails***, ***BuyTickets***, and ***Withdrawals***.

BoxOfficeTheater Component

Like before, we make a call to ***Movie.at(props.query.movie)*** to get the correct instance. With this, we render a page that lets the public watch the correct movie trailer. We also use the instance to call ***getAudienceMembers()*** to retrieve the array of audience members. This lets us check if the viewer has the necessary permission to watch the actual movie. We also pass the array to the ***AudienceMembers*** component for rendering.

8. Conclusions and Future scope

Having a traditional web development background, we made the incorrect assumption that what a typical server/database can handle, the EVM/blockchain too. It was quite frustrating when we kept hitting the block gas limit during the first few iterations of the project!

The lesson for us is to minimize and simplify our data structures and take advantage of event logs as a cheap form of storage. We also have a good reason now to invest in Vyper and LLL, which can reduce our binaries up to 40%! If not mindful, deployment costs for the owner and transaction fees for the users can really add up to quite a paywall!

This platform still needs a lot of polishing! For example, the users should receive precise input validation, not just getting whatever Metamask throws. UI functionalities for the admin are missing! Pagination of active film projects, withdrawal history, and audience members would be more user-friendly. Also, more tooltips to explain what each data element means would lower the learning curve and ease onboarding. And of course, frontend components should have corresponding unit tests of their own to decrease technical debt and make continuous integration possible from end-to-end.This is the place where we had faced a lot of issues with the dependencies especially while linking the front end to the back end through web3js.

Also actual movie uploading can also be done into IPFS (where the max uploading limit is upto 3GB), but network is an issue while uploading such GBs of movies. With the advent of 5G technology in the near future we believe this shouldnt be an issue.