

No-code SCaaS MVP

Prototype MVP for a No-Code Web3 Smart Contract as a Service Platform on
Ethereum.

TONY DATTOLO
tdattolo@iu.edu
CSCI-Y 391: IND SYS DEV
Fall 2021
Advisor: Professor Christina
Chung

CONTENTS

Project Details.....	2
Abstract	2
Core Functionality.....	2
Tech Stack	21
Development Environment and Process	21
VSCode, Ubuntu, pipenv, npm.....	22
Gitflow Workflow	24
Backend	25
Frontend	26
History	26
Project Inspiration	27
Tools	28
Design Patterns.....	28
REST MVT pattern: Django Rest Framework	28
Finite State Machine pattern: Redux.....	32
Persistence Pattern: Redux-Persist.....	32
Factory Pattern: Redux Toolkit-Query	33
Caching and cache invalidation.	35
Development Flow In Practice.....	36
References, Tutorials, Etc.	37

PROJECT DETAILS

ABSTRACT

Prototype MVP for a No-Code Web3 Smart Contract as a Service Platform on Ethereum.

This is a two-semester project. The objective of the first semester is to build the most basic MVP with all the necessary components. A user will be able to register for an account, link wallets to their account, create contracts, invite other users to participate on the contract. The backend will cryptographically verify that users in fact want to construct a contract, and it will then dynamically take the data entered for wallets, clauses, and terms of payout, and generate two things. The first is what is known as a Ricardian contract. It is a human readable version of an Ethereum smart contract, that can be used to settle legal disputes as a back stop, in the same way a traditional dumb contract would be able to in a court of law. This will be generated in the form of a PDF. The second thing generated, is the actual Ethereum smart contract itself. There will be templates those users can select from, which will then be instantiated by the backend with the data specified by the user for their specific instance. Users will then be presented with a launch button, that will require users to verify the intended actions cryptographically once again. Once that step is complete, the backend will launch the contract, and deliver a copy of the Ricardian contract PDF to all involved parties. The contract will then be listed as live on the site, and users will be able to view its details on etherscan.

During the second semester, there will be an attempt to add real world templates, improve UX flow of the contract creation and verification flow, launch contracts to the actual Ethereum blockchain versus a ganache local Ethereum blockchain, and any other steps necessary to move from a basic prototype to a functioning product that users can conduct contracts on. A few examples may be, a marriage contract, an escrow contract for fantasy football leagues, and a sports player's contract with a team with variable payments for performance incentives.

CORE FUNCTIONALITY

1. Basic navigation and UX suitable for MVP version 0 level project
2. Traditional email registration, signup, activation, reset password, etc
3. Dashboard to set account information
4. Connect to users MetaMask Wallet on frontend and interact with it for transactions and signing messages to verify ownership on backend
5. Backend provides logic to validate user ownership over a wallet through signing a unique nonce and verifying with the given details using the Ethereum and eth-utils libraries in a cryptographic manner
6. Allow users to link multiple wallets to the same account. For example, if the user is an NBA team, they may choose to setup a wallet for each player with a contract
7. Contract creation
8. Contract verification before launch
9. Generate PDFs of Ricardian contract
10. A templating system for smart contracts
11. Backend to instantiate templated smart contracts with details provided by users
12. Backend to launch contracts to the local ganache blockchain
13. Contracts and transactions viewable on-chain

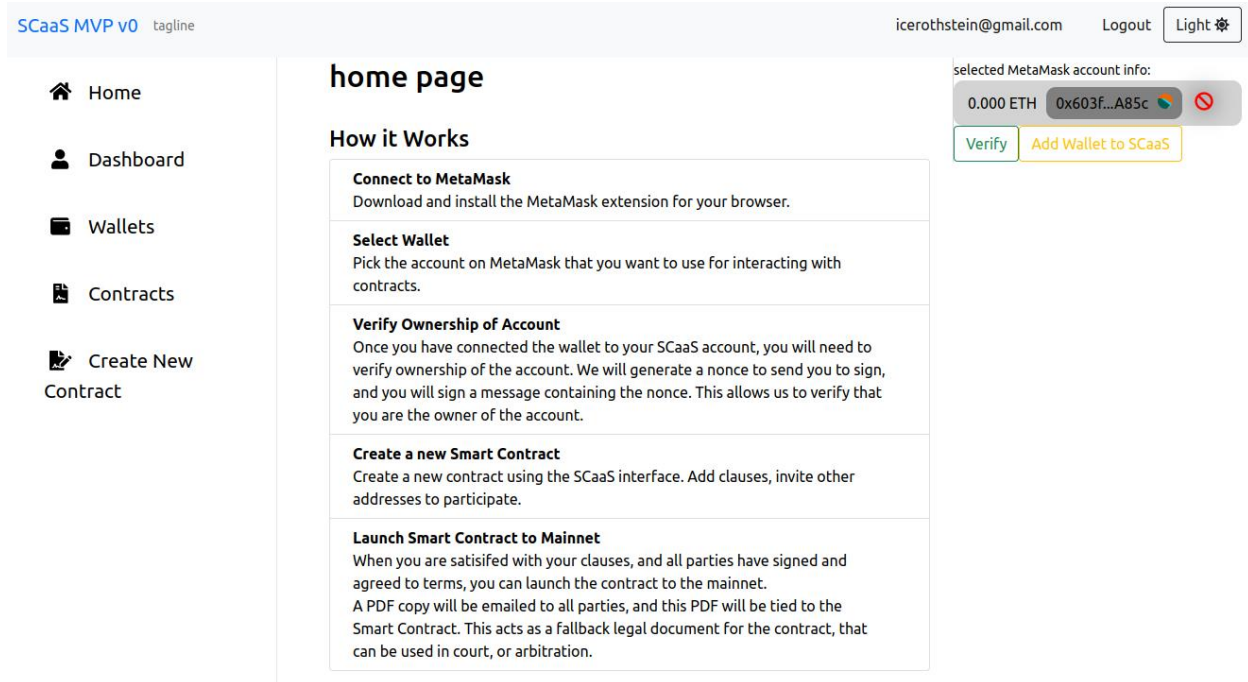


Figure 1: page layout

Signup Page

Email Address

Password

Retype Password

Submit

Already have an activated account? [Login Here](#)

Forget to activate? [Resend activation email](#)

Figure 2: signup

Login Page

Email Address

Password

[Forgot Password?](#)

☐ Check me out

Login

Login With Google

Don't have an account yet? [Sign up here](#)

Figure 3: Login

Request Password Reset :

Email Address

Reset

Don't have an account yet? [Sign up here](#)

Figure 4: password reset

Account Dashboard

Edit Profile

companyRep

bio:

hbomax

first name:

Shiv

first name:

Roy

[Card Link](#) [Another Link](#)

Wallets connected to icerothstein@gmail.com

Wallet: 0x603f5904E5e40e7adf1dCd9A486a3392D5b5A85c

Linked to Site at: 11/29/2021, 12:11:58 AM

Verified: ✓

Name:

Description:

Figure 5: account dashboard

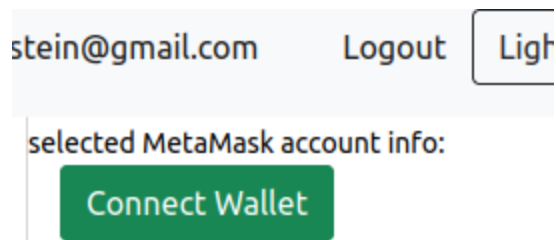


Figure 6: connect wallet button (unconnected)

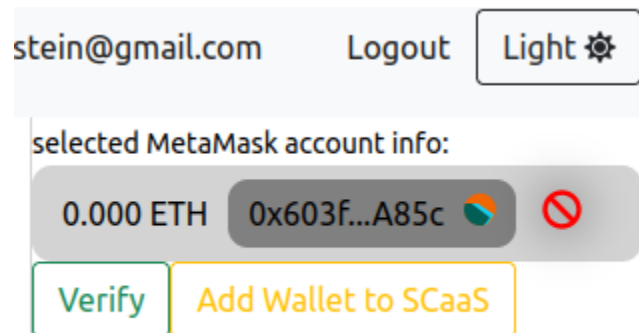


Figure 7: connected wallet

Wallets

Wallets connected to icerothstein@gmail.com

Wallet: 0x603f5904E5e40e7adf1dCd9A486a3392D5b5A85c
Linked to Site at: 11/29/2021, 12:11:58 AM
Verified: ✓
Name:
Description:
Wallet: 0x2C52E8cBF727cEB6585c9aCaBF9dbb7C64477714
Linked to Site at: 12/15/2021, 12:56:54 AM
Verified: ✗
Name:
Description:

Figure 8: unverified wallet added

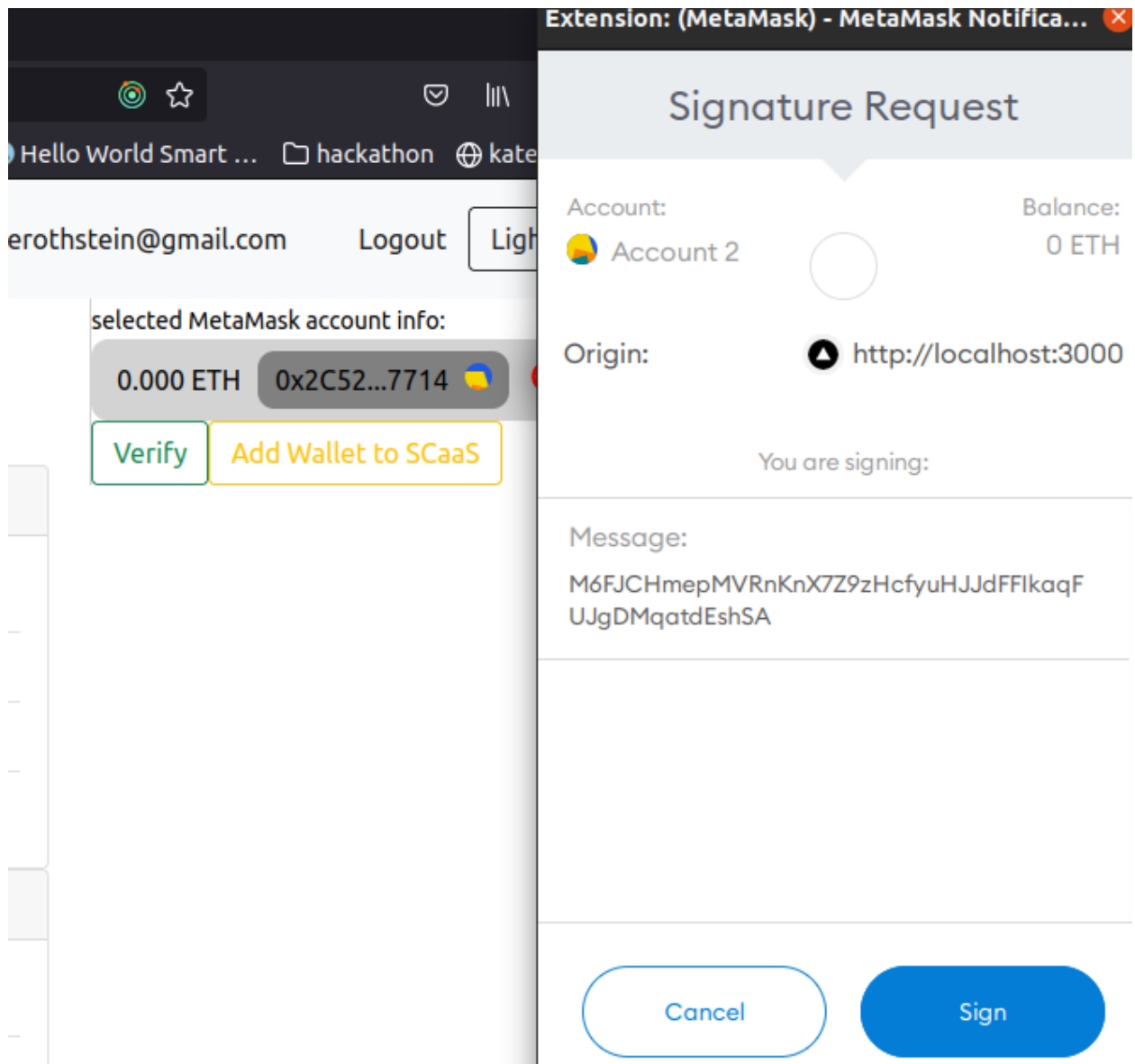


Figure 9: wallet ownership confirmation via backend nonce

Wallets

Wallets connected to icerothstein@gmail.com

Wallet: 0x603f5904E5e40e7adf1dCd9A486a3392D5b5A85c
Linked to Site at: 11/29/2021, 12:11:58 AM
Verified: ✓
Name:
Description:
Wallet: 0x2C52E8cBF727cEB6585c9aCaBF9dbb7C64477714
Linked to Site at: 12/15/2021, 12:56:54 AM
Verified: ✓
Name:
Description:

Figure 10: new wallet verified, multiple wallets added

Contracts

Draft	Pending	Live
Contract: 3rd		View
Created by: icerothstein@gmail.com		
Created at: 11/29/2021, 1:38:21 AM		
Description: lets rock aganeeee		
Status: draft		
Last Updated: 11/29/2021, 1:38:21 AM		
Contract: wager between 2 parties		View
Created by: icerothstein@gmail.com		
Created at: 12/1/2021, 2:50:28 PM		
Description: a wagers that team xyz will beat b		
Status: draft		
Last Updated: 12/8/2021, 4:57:38 PM		

Figure 11: contracts list view

Contract name: wager between 2 parties

Preview PDF

Launch

Created by: icerothstein@gmail.com

Created at: 12/1/2021, 2:50:28 PM

Description: a wagers that team xyz will beat b

Status: draft

Contract Parties

Add Party

icerothstein@gmail.com	sender	Delete Party
withProfile@gmail.com	receiver	Delete Party

Contract clauses

Add Clause

Clause: d896ec9a-1416-440b-b9fc-928e4acf6a31

Delete

if team A beats team B by x amount of points, then party A will pay party B the agreed upon amount

Clause: a99a5661-d85a-4b3a-82a0-ccfedcd19178

Delete

if team x loses, party A pays party B y amount

Edit

Delete

Figure 12: contracts detail view

[Contract Detail](#) / Add Party

Add party to contract: 3725fc58-437b-450f-96e6-105f50a35494

Enter new party email:

lawyer@example.com

Send an accompanying message:

adding receivers lawyer to contract to view clauses

Select role:

lawyer

Add party

Figure 13: add party to contract

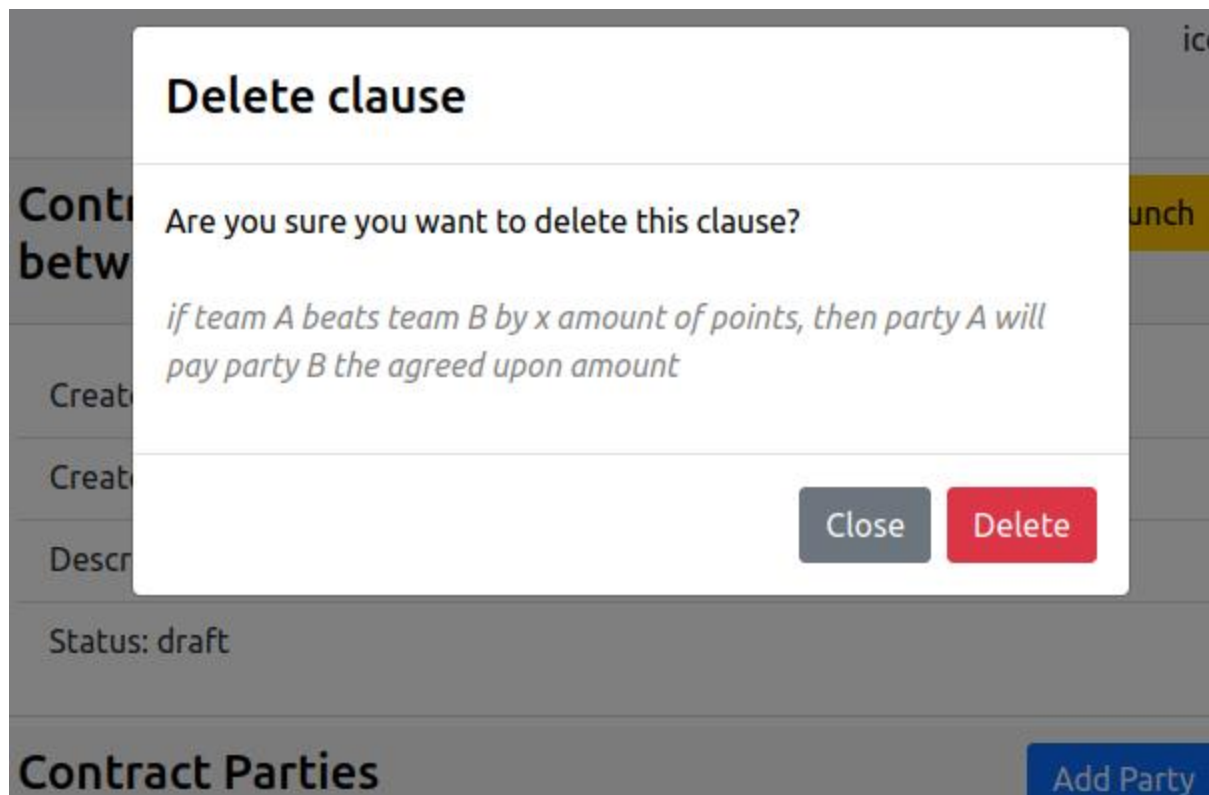


Figure 14: deletion confirmation for clauses and parties

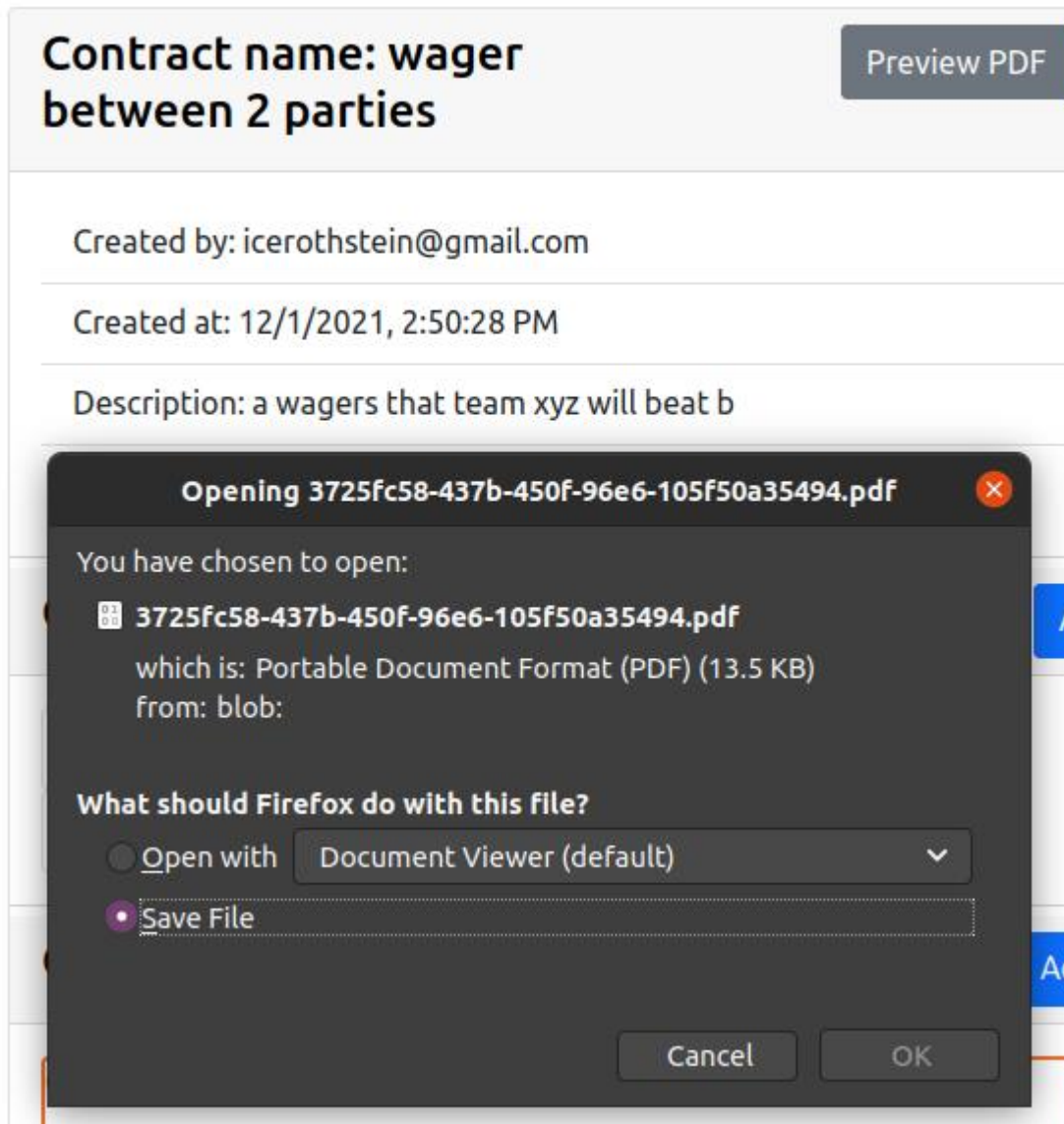


Figure 15: ricardian PDF contract generation

1 of 1

Document
3725fc58-437b-450f-96e6-105f50a35494.pdf

51.8%

Contract 1
Parties 1
Clauses 1

Contract

Name: wager between 2 parties

Description: a wagers that team xyz will beat b

Created At: 01/12/2021

Created By: icerothstein@gmail.com

Parties

Email	Role	Description
icerothstein@gmail.com	sender	
withProfile@gmail.com	receiver	

Clauses

id	content	Created at	
d896ec9a-1416-440b-b9fc-928e4acf6a31	if team A beats team B by x amount of points, then party A will pay party B the agreed upon amount	15.12.2021	#F06F2E
a99a5661-d85a-4b3a-82a0-ccfedcd19178	if team x loses, party A pays party B y amount	15.12.2021	#AE6239

Figure 16: contract details

Success!

The file has been generated.

Contract name: wager between 2 parties

Created by: icerothstein@gmail.com

Preview PDF

Launch

Figure 17: launch contract

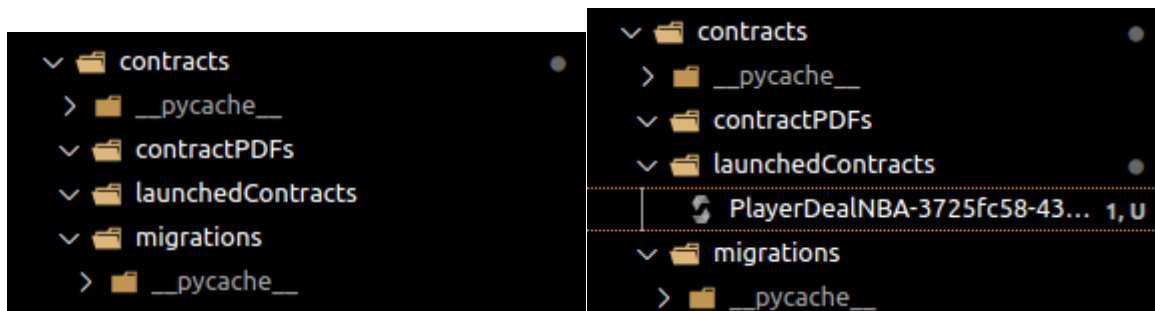


Figure 18: contract instantiated on backend

```

Team public team;
Player public player;
DealTerms public terms;
PerformanceBonus[] public bonuses;

function PlayerDealNBA(AggregatorV3Interface _priceFeed
|   priceFeed = _priceFeed;
}

// constructor only called when contract is created
constructor () {
    // set the price feed for ethereum. maybe coinbase
    priceFeed = AggregatorV3Interface(0xAc559F25B161917);
    team = new Team();
    player = new Player();
    terms = new DealTerms();
    bonuses = new PerformanceBonus[];
}

// function getPrice(string memory _playerName) public
//     return priceFeed.getAggregatedValue(_playerName);
// }

mapping (bool => uint) performanceBonus;
event WeeklySalaryPaid(address indexed player, uint256 salary);
string public senderEmail = "icerothstein@gmail.com";

```

Figure 19: instantiated contract with dynamic information generated automatically

```

1  pragma solidity ^0.8.10 <0.9.0;
2
3
4  contract WagerEscrow {
5
6      address public partyA;
7      address public partyB;
8      address public arbiterOracle;
9      bool public WINNER_A = false;
10     bool public WINNER_B = false;
11     bool public TIE = false;
12     bool public CANCELLED = false;
13
14     uint public wagerAmount;
15
16
17     // function to have msg.sender set the partyA
18     function setPartyA(address _partyA) public {
19         partyA = _partyA;
20     }
21     // function to have msg.sender set the partyB
22     function setPartyB(address _partyB) public {
23         partyB = _partyB;
24     }
25
26     // function to have msg.sender set the arbiterOracle
27     function setArbiterOracle(address _arbiterOracle) pub
28     |   arbiterOracle = _arbiterOracle;
29     }
30
31     // function to have msg.sender set the wagerAmount
32     function setWagerAmount(uint _wagerAmount) public {
33         wagerAmount = _wagerAmount;
34     }

```

Figure 20: example of non-instantiated smart contract

```
5 |
6 | address public partyA;

bash:: ~81x24
~#> ganache-cli --deterministic
ganache v7.0.0-beta.1 (@ganache/cli: 0.1.1-beta.1, @ganache/core: 0.1.1-beta.1)
Starting RPC server

Available Accounts
=====
(0) 0x90F8bf6A479f320ead074411a4B0e7944Ea8c9C1 (1000 ETH)
(1) 0xFFcf8FDEE72ac11b5c542428B35EEF5769C409f0 (1000 ETH)
(2) 0x22d491Bde2303f2f43325b2108D26f1eAbA1e32b (1000 ETH)
(3) 0xE11BA2b4D45Eaed5996Cd0823791E0C93114882d (1000 ETH)
(4) 0xd03ea8624C8C5987235048901fB614fDcA89b117 (1000 ETH)
(5) 0x95cED938F7991cd0dFcb48F0a06a40FA1aF46EBC (1000 ETH)
(6) 0x3E5e9111Ae8eB78Fe1CC3bb8915d5D461F3Ef9A9 (1000 ETH)
(7) 0x28a8746e75304c0780E011BEed21C72cd78cd535E (1000 ETH)
(8) 0xACa94ef8bD5ffEE41947b4585a84BdA5a3d3DA6E (1000 ETH)
(9) 0x1dF62f291b2E969fB0849d99D9Ce41e2F137006e (1000 ETH)

Private Keys
=====
(0) 0x4f3edf983ac636a65a842ce7c78d9aa706d3b113bce9c46f30d7d21715b23b1d
(1) 0x6cbcd15c793ce57650b9877cf6fa156fbef513c4e6134f022a85b1ffdd59b2a1
(2) 0x6370fd033278c143179d81c5526140625662b8daa446c22ee2d73db3707e620c
(3) 0x646f1ce2fdad0e6deeeb5c7e8e5543bdde65e86029e2fd9fc169899c440a7913
(4) 0xadd53f9a7e588d003326d1cbf9e4a43c061aadd9bc938c843a79e7b4fd2ad743
32 | function setWagerAmount(uint _wagerAmount) public {
33 |     |   wagerAmount = _wagerAmount;
34 | }

PROBLEMS (2) OUTPUT DEBUG CONSOLE TERMINAL

Generating build data...
- SimplePaymentChannel
- WagerEscrow

BrownieProject is the active project.
Attached to local RPC client listening at '127.0.0.1:8545'...

Running 'scripts/deploy.py::main'...
Deploying Wagers contract...
partyA=<Account '0x90F8bf6A479f320ead074411a4B0e7944Ea8c9C1'>
partyB=<Account '0xFFcf8FDEE72ac11b5c542428B35EEF5769C409f0'>
arbiterOracle=<Account '0x22d491Bde2303f2f43325b2108D26f1eAbA1e32b'>
~#> brownie > feature-generateContract # ~1 ?1
02:11:19
1m 9s 4ms
```

Figure 21: brownie using our deterministic ganache-cli chain accounts to deploy and test contracts

Ganache

ACCOUNTS

BLOCKS

TRANSACTIONS

CONTRACTS

EVENTS

LOGS

SEARCH FOR BLOCK NUMBERS OR TX HASHES

CURRENT BLOCK
2

GAS PRICE
20000000000

GAS LIMIT
6721975

HARDFORK
MUIRGLACIER

NETWORK ID
1337

RPC SERVER
HTTP://127.0.0.1:7545

MINING STATUS
AUTOMINING

WORKSPACE
MATURE-NOSE

SWITCH

TX HASH

0x740211b0a1def5c420c120fade7dc6e39c9ebc80d261b912d18fa56e89cbebab

CONTRACT CALL

FROM ADDRESS

0x407f715EE3C2b860cb135dffF62C2C748d68c23B

TO CONTRACT ADDRESS

0x4837FFb8eA14A1db32B6DB3d4df179Fc51d736F9

GAS USED

29602

VALUE

0

TX HASH

0xe377e4c980b1db1cac5ed1fecfef8165cacf9d6d1411dbdfb49d25f1f2d6e34a

CONTRACT CREATION

FROM ADDRESS

0x407f715EE3C2b860cb135dffF62C2C748d68c23B

CREATED CONTRACT ADDRESS

0x4837FFb8eA14A1db32B6DB3d4df179Fc51d736F9

GAS USED

375254

VALUE

0

Figure 22: ganache gui blockchain explorer showing how contracts can be viewed similar to etherscan on live ethereum chains

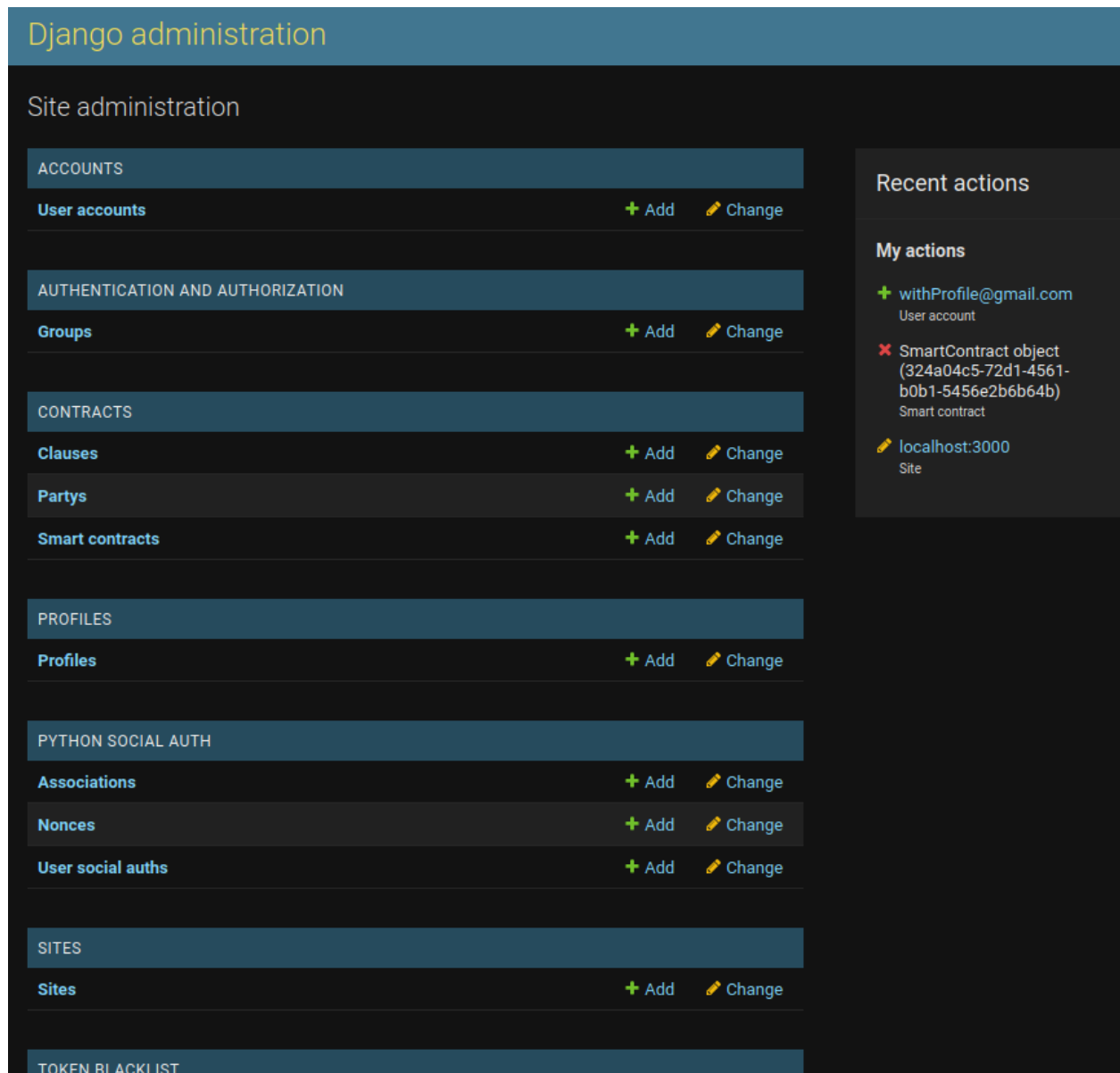


Figure 23: backend admin panel for quick gui control over critical functions like banning adverse users or exploitative contracts

SCaaS API ^{v0}

[Base URL: 127.0.0.1:8000/]

<http://127.0.0.1:8000/swagger/?format=openapi>

SCaaS mvp api layer

[Terms of service](#)

[Contact the developer](#)

[BSD License](#)

Schemes

HTTP

Filter by tag

auth

POST /auth/jwt/create/

POST /auth/jwt/refresh/

POST /auth/jwt/verify/

Figure 24: API layer gui explorer to allow for testing and JSON structure viewing for FE engineers

TECH STACK

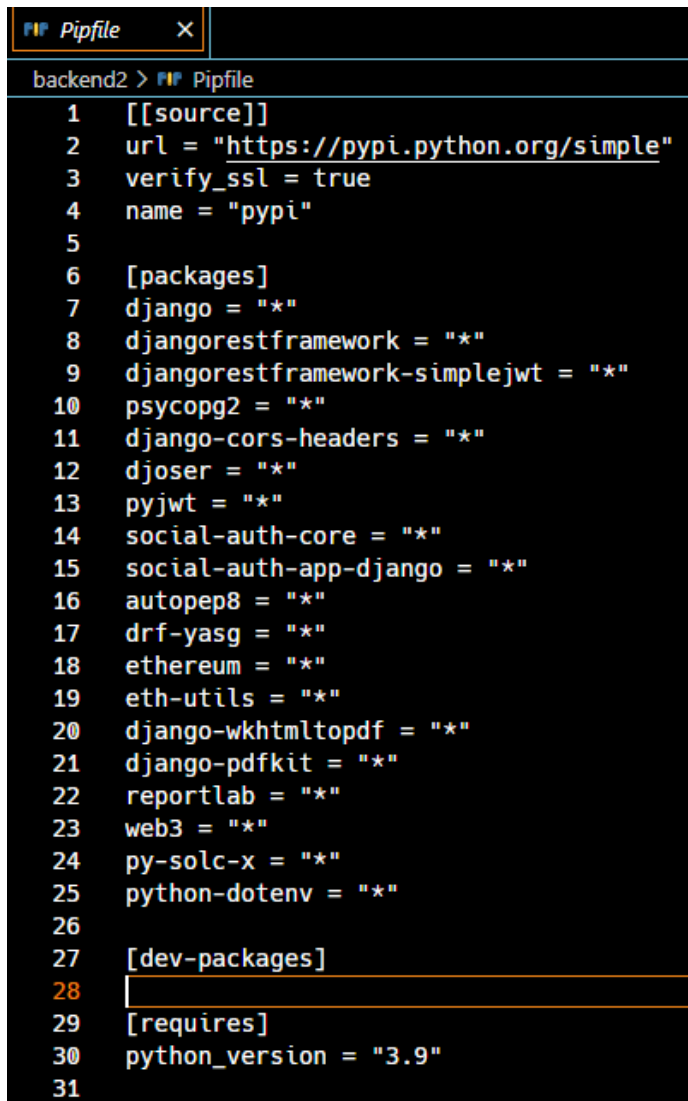
DEVELOPMENT ENVIRONMENT AND PROCESS

VSCODE, UBUNTU, PIPENV, NPM

The IDE used for this project was VSCode. The extension library and customization functionality ensures maximization of our productivity. Keybindings, native or extension support for every necessary language, framework, tool, etc is available. There is a robust directory manager sidebar that allows us view the codebase directory structure as it scales. When dealing with React for example, there are many approaches to directory structure [1]. This might seem unimportant, but as the project grows sufficiently large over time, the directory structure becomes more important. Visibility into that aspect, allows our developers to keep this front of mind as necessary. If there starts to be dozens of custom components, we might want to break component directory into different features, or pages, or type of component, etc.

The debian based Linux distro, Ubuntu, was used for the entirety of the project. Linux improves developer productivity by allowing for more direct control of over the development environment, customization, and first class integration support. For example, built-in PostgreSQL support, or installing compiled binaries for PDF generation libraries, or installation and management of environments like NodeJS.

Pipenv was used for module management on the backend python environment.



```
1  [[source]]
2  url = "https://pypi.python.org/simple"
3  verify_ssl = true
4  name = "pypi"
5
6  [packages]
7  django = "*"
8  django-rest-framework = "*"
9  django-rest-framework-simplejwt = "*"
10 psycpg2 = "*"
11 django-cors-headers = "*"
12 djoser = "*"
13 pyjwt = "*"
14 social-auth-core = "*"
15 social-auth-app-django = "*"
16 autopep8 = "*"
17 drf-yasg = "*"
18 ethereum = "*"
19 eth-utils = "*"
20 django-wkhtmltopdf = "*"
21 django-pdfkit = "*"
22 reportlab = "*"
23 web3 = "*"
24 py-solc-x = "*"
25 python-dotenv = "*"
26
27 [dev-packages]
28
29 [requires]
30 python_version = "3.9"
31
```

Figure 25: python modules used in pipfile for pipenv

Npm was used for package management on the frontend, as well as installing certain cli tools like ganache-cli


```
frontend2 > npm package.json > ...
5  "scripts": {
6    "dev": "next dev",
7    "build": "next build",
8    "start": "next start",
9    "lint": "next lint"
10 },
11 "dependencies": {
12   "@fortawesome/fontawesome-svg-core": "^1.2.36",
13   "@fortawesome/free-regular-svg-icons": "^5.15.4",
14   "@fortawesome/free-solid-svg-icons": "^5.15.4",
15   "@fortawesome/react-fontawesome": "^0.1.15",
16   "@metamask/detect-provider": "^1.2.0",
17   "@metamask/jazzicon": "^2.0.0",
18   "@reduxjs/toolkit": "^1.6.1",
19   "@usedapp/core": "^0.5.4",
20   "axios": "^0.24.0",
21   "bootstrap": "^5.1.1",
22   "ethers": "^5.5.1",
23   "formik": "^2.2.9",
24   "js-file-download": "^0.4.12",
25   "next": "^11.1.2",
26   "react": "17.0.2",
27   "react-bootstrap": "^1.6.3",
28   "react-dom": "17.0.2",
29   "react-redux": "^7.2.5",
30   "redux": "^4.1.1",
31   "redux-devtools-extension": "^2.13.9",
32   "redux-persist": "^6.0.0",
33   "redux-thunk": "^2.3.0",
34   "sass": "^1.40.0",
35   "yup": "^0.32.11"
36 }
```

Figure 26: npm packages used on frontend

GITFLOW WORKFLOW

This project used Gitflow Workflow, also commonly known as feature-branching, to utilize the full capabilities of git. The project main branch is reserved for deployment, where testing infrastructure and a continuous integration/continuous deployment methodology, such as GitHub actions, can be employed. This is to ensure maximum uptime of an in-production app, and ensure we minimize bugs going into production. The develop branch is then used for handling the job of merging new features into the existing codebase. Develop in its default state maintains an exact copy of the main branch, so that merge conflicts and failing tests can be handled before being pushed to production. It also allows a fresh state for developers to branch off. It also allows for hotfix branches which can test bugs found in production to make sure they're patched before merging into main. You can also designate separate release branches to leave snapshots of official launches to production on the repo.

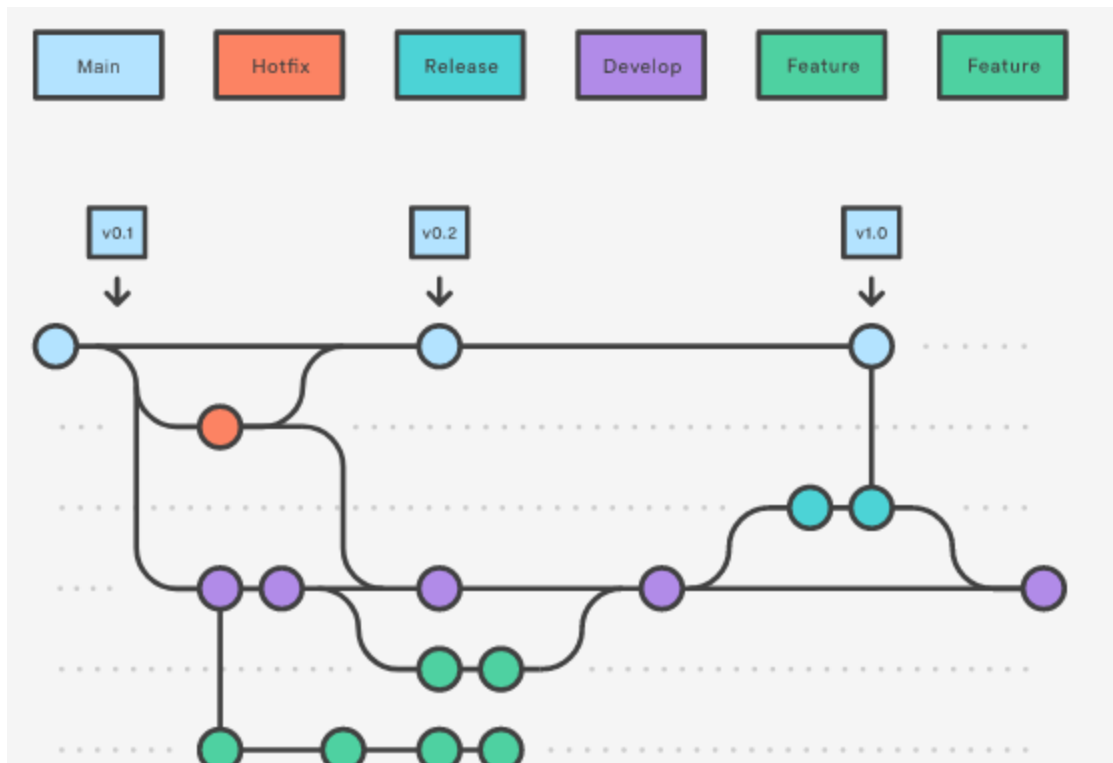


Figure 27: gitflow workflow diagram

The screenshot shows a split terminal window. The left pane shows the backend running in a terminal with the following commands and output:

```

/home/tony/REPOs/web3Social/backend2/.venv/bin/activate
backend2 $ feature-generatePDF # ? ?1
/home/tony/REPOs/web3Social/backend2/.venv/bin/activate
backend2 $ feature-generatePDF # ? ?1
python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
December 03, 2021 - 09:41:35
Django version 3.2.9, using settings 'backendAPI.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.

```

The right pane shows the frontend running in a terminal with the following commands and output:

```

~#() cd frontend2
frontend2 $ feature-generatePDF # ? ?1
~#() npm run dev

> frontend@0.1.0 dev
> next dev

ready - started server on 0.0.0.0:3000, url: http://localhost:3000
info - Using webpack 5. Reason: Enabled by default https://next.js.org/docs/messages/webpack5
warn - Detected next.config.js, no exported configuration
ocs/messages/empty-configuration
event - compiled successfully

```

Figure 28: split terminals showing frontend/backend running concurrently in the designated feature-branch

BACKEND

- Django <https://docs.djangoproject.com/en/4.0/>
- Django Rest Framework <https://www.django-rest-framework.org/>
- Web3.py <https://web3py.readthedocs.io/en/stable/>

- Brownie <https://eth-brownie.readthedocs.io/en/stable/>
- Eth-utils <https://eth-utils.readthedocs.io/en/latest/>
- Py-solc-x <https://github.com/iamdefinitelyahuman/py-solc-x>
- Swagger <https://swagger.io/tools/swagger-ui/>
- Wkhtmltopdf <https://github.com/wkhtmltopdf/wkhtmltopdf>
- Djoser <https://djoser.readthedocs.io/en/latest/>

FRONTEND

- NodeJS <https://nodejs.org/en/>
- NextJS <https://nextjs.org/docs/getting-started>
- Scss modules <https://nextjs.org/docs/advanced-features/customizing-postcss-config#css-modules>
- Redux <https://redux.js.org/usage/index>
- Redux Toolkit <https://redux-toolkit.js.org/usage/usage-guide>
- RTK-Query <https://redux-toolkit.js.org/rtk-query/overview>
- Redux Persist <https://github.com/rt2zz/redux-persist>
- React-Bootstrap <https://react-bootstrap.github.io/components/alerts>
- useDapp <https://usedapp.readthedocs.io/en/latest/index.html>
- MetaMask <https://docs.metamask.io/guide/>
- Fontawesome <https://fontawesome.com/v5.15/how-to-use/on-the-web/using-with/react>
- Axios <https://axios-http.com/docs/intro>

HISTORY

- Blockchains, the technology underlying the entire Web3 movement, is no accident. It's the culmination of 80 years of research and advancements in cryptography, data structures and algorithms, and distributed systems. In the 60s, computer networks and ARPANET pioneered the field of distributed systems. In the 1970s, there were two significant advancements in the field of cryptography and data structures that are relevant to this project. The first was public key cryptography pioneered by Diffie-Hellman, and the second was the Merkle Tree data structure. In the 80s, work was done on the Byzantine Generals Problem in distributed systems, and elliptic curve cryptography. In the 90s, we began to see the emergence of utilizing these technologies to begin to solve real world problems. Digicash and Bit Gold were early attempts at cryptocurrency, three software engineers came up with a way to tamper proof timestamps with Merkle trees, and the peer-to-peer network Napster showed there was a way to distribute files globally. In 2009, an author(s) under the pseudonym Satoshi Nakamoto released a whitepaper outlining a new form of digital currency that solves both the Byzantine Generals Problem and the double-spend problem utilizing elliptic key cryptography, Merkle trees, and proof of work. Finally, in 2014, Vitalik Buterin and a coalition of developers launched Ethereum. The first Turing Complete Blockchain, and with it the invention of smart contracts and the solidity language. Which brings us to today.

- The technology is very early. You often find yourself asking questions on forums, and it's for things that have never been done before and are just theory. In theory, because we now have a Turing complete interface for a cryptographic protocol capable of securely transferring currency, we can build literally anything. We can prove digital ownership of assets, we can prove identities and medical records, we can automatically transfer funds on real world events, we can provide liquidity for formerly illiquid assets, we can prove ownership, we can distribute

rewards, we can have built in incentive mechanisms into governance and operations of organizations. The list goes on and on and on.

PROJECT INSPIRATION

In the past 5 years, the broader ecosystem has largely focused on infrastructure and tooling. Exchanges, frameworks, languages, new blockchains, hosting services, staking services, wallet services, and on and on.

The inspiration for this project came from a desire to begin to solve more problems for users utilizing smart contracts at the B2B and B2C level. To begin to move from dev tooling and infra, to the mainstream. The first use case in my eyes seems rather obvious. Turn dumb contracts, into smart contracts. It sounds simple but it's not easy task. Contracts have many parties and clauses, are often expansive, broad, and full of jargon. They're just words on a paper, but our legislative branch ensures these words have enforceable power, and thus trigger events for clauses. They are drafted by expert teams of lawyers who often tout on their marketing materials the combined years of experience of all the practicing partners at the firm. The law itself is constantly evolving and the power behind clauses can either be nullified through a courts ruling, or new laws can be introduced. Often when we discussion innovation and disruption, there are statements about things being completely replaced. In my eyes, this realm we are entering often touches upon the human element. Something happens - between human parties not just computers – in the real world and that has some agreed upon effect that needs to be accounted for. As such, I do not see things like lawyers being replaced by technology. I see a real need for their expertise, and rather than try to displace them, I seek to build tooling at the business and consumer level, to enable these users to solve their problems more efficiently, or inline with the times in a way that ensures they are best prepared to tackle the future.

Let's take an example. A sports contract. Party A, the team, wants to sign a new player, Party B. It's a great first example, because there's just two parties involved in terms of the transfer of value. There are also lawyers and managers, who may also need to be paid, but at its core, the triggers for the exchange of value for services rendered are between two parties. There are clauses in the contract. There may be a fixed amount of compensation over time, but there may also be other variable conditions based on real world events. Performance bonuses or something like number of games or minutes played. This variable portion is particularly interesting because it adds another level of complexity to our smart contract. There is now a physical real world action that needs to occur, in order for a clause to be triggered. So, how do we inform the digital of the physical without human intervention. Should we just have the players manager phone up the club to remind them, or can we partner with the league, access an official data API, and construct an Oracle that feeds information to our smart contracts, triggering clauses automatically. I think that's doable, and that's what we're trying to do here.

So what about the real world, what about the courts and the lawyers. What happens if something goes wrong? Well, despite all our technological advances, humanity remains chaotic and unpredictable, and there needs to be some stop gap. We still need our courts and lawyers to be able to understand and interpret things we are doing in code. Ricardian contracts are designed to be both human and machine readable. We want not only the Ethereum blockchain to understand what we want to do, but the local judge who to decipher how to proceed if something goes wrong. That really ties into the core of our service. This is not just for the digital realm. We are bridging the digital and the physical worlds using a mapping process that is easily deciphered by the existing legal system and non-technical users.

TOOLS

useDapp has been selected to interface with users wallets. It is an abstraction layer built off of the most popular frontend web3 library, ethersJS. It provides react hooks that fit nicely with the rest of our approach. We're using hook abstractions all over, so it just makes sense to keep it consistent. It also has utilities for common functions like truncating an address for display or checking address matches on signatures.

On the backend, we are using web3.py to interact directly with the contracts, templates, instantiate templated contracts dynamically, and launch.

Ganache is our local Ethereum blockchain used for testing rapidly in development. Ropsten is then used as we move towards production, most closely resembling mainnet. This is where the contracts can be inspected on etherscan in the same manner they would be on mainnet, and the contract can be audited prior to mainnet launch. Brownie will be used to make interacting with contracts easier, such as deployment, transactions, testing, etc.

DESIGN PATTERNS

REST MVT PATTERN: DJANGO REST FRAMEWORK

The MVT design pattern is what drives Django, similar to the original MVC pattern. In this case, it's considered a headless MVT pattern, because we're using a REST API implementation to interface with our frontend, rather than serving templates or react within the actual django backend itself. Model refers to our data model, which is located within the models.py file of every app. Models can have field validations, defaults, define roles and permissions, and more. They interface directly with whatever database is set to the default in settings. Out of the box it starts you off with sqlite3 which is enough to get started, but PostgreSQL is a first class citizen in Django and all features are supported. The models define the structure of the data in the database, which then gets called by many operations throughout the application. For example, when making a http request to the backend, the user passes in some data about a specific object, such as the primary key of a smart contract. The query then takes that object id, and uses it to grab the object from the database using the model.

```
10 class SmartContract(models.Model):
11     """
12     This class represents the SmartContract model.
13     """
14     id = models.UUIDField(primary_key=True, default=uuid4, editable=False)
15     owner = models.ForeignKey(USER, on_delete=CASCADE)
16     name = models.CharField(max_length=100)
17     description = models.TextField(blank=True)
18
```

Figure 29: Smart Contract Model

```

45         smartContract = SmartContract.objects.create(
46             owner=owner,
47             name=name,
48             description=description
49         )
50         smartContract.save()
51         party = Party.objects.create(
52             contract=smartContract,
53             party=owner,
54             role='sender'
55         )

```

Figure 30: example of a model being utilized inside a view to get, filter, create, etc.

In the REST implementation of the MVT pattern, when our frontend sends requests to the backend Django server, it comes in the form of JSON objects. As such, we need some way to process the flow of information in a way that can be used by our Django models. This process is called deserialization, and luckily for us, Django REST Framework comes with a built-in serializer class that can be extended to serialize and deserialize all of our models. It also allows for returning wanted fields from ForeignKey models attached to the model. For example, our base User class handles all of our traditional Web2 JWT based account CRUD and authentication functionality. A Party to a contract uses the User model as a ForeignKey, therefore the Party itself does not actually contain the email for that party. If we want to see it within the contract, we can simply define a new field referencing the User model from within the defined ForeignKey of the Party. In this case, our party attribute references the user model, and as such we set the source of our custom serializer field to source='party.email'

```

3  from rest_framework import serializers
4
5  from .models import SmartContract, Clause, Party
6
7  class SmartContractSerializer(serializers.ModelSerializer):
8      owner = serializers.ReadOnlyField(source='owner.email')
9      class Meta:
10         model = SmartContract
11         fields = '__all__'
12
13
14  class ClauseSerializer(serializers.ModelSerializer):
15      class Meta:
16         model = Clause
17         fields = '__all__'
18
19  class PartySerializer(serializers.ModelSerializer):
20      partyEmail = serializers.ReadOnlyField(source='party.email')
21      class Meta:
22         model = Party
23         fields = ('id', 'partyEmail', 'description', 'role',)
24         # fields = '__all__'

```

Figure 31: serializers with custom fields and model fields

Now that the Model portion of MVT is considered, we have views.py to define our business logic and api endpoints. Django has built in support for a wide range of http requests types, abstractions, inheritable generic classes, permissions, authentications, etc. Anything you would need to control the flow in or out of your application for whatever reason, there is a built-in answer.

```

23 class CreateSmartContractView(APIView):
24     """
25     Create a smart contract
26     """
27     permission_classes = [permissions.IsAuthenticated,]
28     authentication_classes = [JWTAuthentication,]
29
30     def post(self, request):
31
32
33         owner = USER.objects.get(email=request.data['owner'])
34         name = request.data['name']
35         description = request.data['description']
36         print(f'{owner=}, {name=}, {description=}')
37
38         if not owner or not name or not description:
39             return Response({"message": "Missing fields"}, status=status.HTTP_400_BAD_REQUEST)
40
41         if SmartContract.objects.filter(owner=owner, name=name).exists():
42             return Response({"message": "Smart contract with that name already exists"})
43
44         try:
45             smartContract = SmartContract.objects.create(
46                 owner=owner,
47                 name=name,
48                 description=description
49             )
50             smartContract.save()
51             party = Party.objects.create(
52                 contract=smartContract,
53                 party=owner,
54                 role='sender'
55             )
56             print(f'{party=}')
57             party.save()
58             return Response(
59                 {"success": "smart contract successfully created"},
60                 status=status.HTTP_201_CREATED
61             )
62         except Exception as e:
63             return Response(
64                 {"message": f"error creating post:{e}"},
65                 status=status.HTTP_500_INTERNAL_SERVER_ERROR)

```

Figure 32: example of an api view with error handling, edge cases, responses, filtering, etc

Once the views are complete, instead of rendering a template, we define a REST API endpoint in our urls.py for the project which can then be called from the frontend with the appropriate variables and authentication status, to request or mutate data.


```

3  from .views import (
4      CreateSmartContractView,
5      SmartContractListViewByOwner,
6      SmartContractDeleteView,
7      GetContractDetailView,
8      AddPartyToSmartContractView,
9      AddClauseToContractView,
10     DeleteClauseFromContractView,
11     GeneratePDFPreviewView
12 )
13
14 urlpatterns = [
15     path('create/', CreateSmartContractView.as_view(), name='create'),
16     path('list/<str:email>/<str:type>/', SmartContractListViewByOwner.as_view(), name='list'),
17     path('delete/<str:id>/', SmartContractDeleteView.as_view(), name='delete'),
18     path('detail/<str:id>/', GetContractDetailView.as_view(), name='detail'),
19     path('add_party/<str:id>/', AddPartyToSmartContractView.as_view(), name='add_party'),
20     path('add_clause/<str:contract_id>/', AddClauseToContractView.as_view(), name='add_clause'),
21     path('delete_clause/<str:contract_id>/', DeleteClauseFromContractView.as_view(), name='delete_clause'),
22     path('generate_pdf/<str:contract_id>/', GeneratePDFPreviewView.as_view(), name='generate_pdf')
23 ]
24

```

Figure 33: example of api endpoints defined for our views

FINITE STATE MACHINE PATTERN: REDUX

Maintaining statefulness on the frontend is no simple task. In large applications you have different states for authenticated users, permissions, groups, data for each interaction. It can get quite complicated. The modern implementation of Redux allows users to separate their data into chunks to make it more manageable to work with. This is done using *slices*. `createSlice` function generates all the boilerplate needed to handle the structure of the data, and make changes. Under the hood it uses the `immer` library, which is based off of the tried-and-true Finite State Machine pattern. This pattern is very important because it allows for time-travel debugging. In a complex app, maybe there are 8 actions on the frontend that occur in the background. The way this works, is at each step, if we open our Redux DevTools Extension, because the data is immutable in this pattern, we can view the changes as they happen with each action. This allows us to pinpoint exactly where something is going wrong, or right.

- <show example of data changes being shown in redux dev tools>

PERSISTENCE PATTERN: REDUX-PERSIST

- While redux handles how to structure the state of our data, what it doesn't do is persist the data across sessions. This is commonly referred to as hydration and rehydration. The `redux-persist` library allows for the persistence of data. For example, without `redux-persist`, when we would navigate away from the web app and

come back, our state would not always be saved. If you think about the modern user experience, users expect this functionality. They don't want to be constantly relogging in, or repeating flows they've already completed. Redux-persist allows for data-persistence even when leaving over time. If you didn't clear your cache, and came back months later, as long as it was within the time period of the JWT refresh token expiration timeline, you'd still be logged in and able to continue on with where you left off. With this we also need to purge the state of our data upon logging out, and luckily this feature is included as well. Rather than manually updating everything, we can simply add a PURGE function to our data mutation on setLogout.

This is not without its cons. There is additional overhead for maintaining the redux store as it pertains to persistence. As new slices and APIs are added to the application, their slice structure, API namespace must be added to the store watcher and blacklist, and the middleware must be concatenated to the default redux middleware in order to maintain functionality for things like onWindowFocus event listener triggers.

- <show additional configuration in store>

FACTORY PATTERN: REDUX TOOLKIT-QUERY

For this project, I opted for a highly decoupled approach to structuring the code base. Thus, it makes sense to adopt a design pattern and abstraction layer to decouple one of the most common use cases: making queries from the frontend to the REST API layer.

When making an API call, we end up repeating the same 4 cases over and over. We need to listen for an error, we need a state for loading, and need a case for successful data returned but empty, and of course successful data returned but not empty. Redux Toolkit Query was built specifically to handle this purpose and provides hooks and variables that are consistent for every query. This ensures that no matter what we're trying to do, each query will follow the same structure, minimizing the code needed to check for edge cases, and keeping the codebase in a consistent style regardless of the app or who is working on it. It also abstracts the query information away in a separate area of the code base, under the slices directory, and specifically within the <name>API.js file for that feature. Inside the component, you can define the data, using namespaces if there are multiple queries in a component, and you can make the async/await call using a one-line asynchronous hook provided by the library. No more 50 lines of logic cluttering every component, this boils it down to around 10 on average, give or take event listeners needed for post query actions outside the data fetching and loading functionality.

```

30     createContract: builder.mutation({
31       query(data) {
32         // console.log(`data: ${data}`)
33         const { owner, name, description, access_token } = data;
34         return {
35           url: "create/",
36           headers: {
37             Authorization: `JWT ${access_token}`,
38             Accept: "application/json",
39             "Content-Type": "application/json",
40           },
41           method: "POST",
42           body: {
43             owner: `${owner}`,
44             name: `${name}`,
45             description: `${description}`,
46           }
47         };
48       },
49       invalidatesTags: ["contracts"],

```

Figure 34: abstracted API call logic inside of createApi RTK-Q function

```

const [createContract, {
  error: createContractError,
  isSuccess: createContractIsSuccess,
  isLoading: createContractIsLoading,
  isError: createContractIsError,
}] = useCreateContractMutation()

const handleNewContractSubmit = async () => {
  try {
    await createContract({owner, name, description, access_token })
  } catch (error) {
    console.log(error)
  }
}

useEffect(() => {
  if (createContractIsSuccess) {
    router.push("/contracts", undefined, { shallow: true })
  }
}, [createContractIsSuccess])

```

Figure 35: RTK-Query hooks/variables in action inside a component

```

94   return (
95     <>
96       {generatePDFfromContractIsError && <Alert variant="danger">{generatePDFfromContractError.message ?? "error gen
97     }
98     {deleteContractIsError && (
99       <Alert variant="danger">
100       | {deleteContractError.message ?? "Error deleting contract"}
101       </Alert>
102     )}
103
104     {contractIsLoading && <Spinner animation="border" variant="primary" />}
105     {contractIsError && contractError.data && <Alert variant="danger">{contractError.data.message}</Alert>}
106     {contractIsSuccess && contractData} && (
107       <>
108         <Card className={`mt-4`} >
109           <Card.Header>
110             <Row>
111               <Col>
112               | <h4>Contract name: {contractData.contract.name}</h4>
113               </Col>
114               <Col>
115                 <Button
116                   variant="primary"
117                   onClick={handleGeneratePDF}
118                   disabled={generatePDFfromContractIsLoading}
119                   style={{ float: "right" }}
120                   download
121                 >
122                 | Preview PDF
123                 </Button>
124               </Col>

```

Figure 36: RTK-Q variables being used to conditionally render a react component using boolean and ternary operations

CACHING AND CACHE INVALIDATION.

Another bonus of using the RTKQ library, is it comes with built in data caching and cache invalidation through the concept of tags. You define a tag within the createApi function of that slice of data, and whenever you want the page to automatically refresh and update something like a list of objects, you simply define on which successful api query you want the tag invalidated, and automatically refetch the data. It's essentially an event listener for our caches. It's a seemingly small feature, but it's critical to the UX functionality of a SPA application. For example, on a social media site, you have a feed of posts, and when you create a new post, you expect the page to display it without reloading the page. This feature makes it appear as though it were magic, but really we're just making an api call, invalidating our cache, and refetching the new list of data with the added post, or in our case contract/clause/party.

```

2
3 export const contractsApi = createApi({
4   reducerPath: "contractsApi",
5   baseQuery: fetchBaseQuery({
6     baseUrl: "http://localhost:8000/contracts/",
7     prepareHeaders: (headers, { getState }) => {
8       headers.set("Accept", "application/json");
9       headers.set("Content-Type", "application/json");
10      return headers;
11    },
12  }),
13  tagTypes: ["contracts", "contractDetail"],
14  endpoints: (builder) => ({
15    getAllContracts: builder.query({
16      query: () => "",
17      providesTags: ["contracts"],
18    }),
19    getDraftContractsByUser: builder.query({
20      query(data) {
21        const { email, access_token } = data
22        return {
23          url: `list/${email}/draft/`,
24          headers: {
25            "Authorization": `JWT ${access_token}`,
26          },
27        }
28      },
29    }),
30    createContract: builder.mutation({
31      query(data) {
32        // console.log(`data: ${data}`)
33        const { owner, name, description, access_token } = data;
34        return {
35          url: "create/",
36          headers: {
37            Authorization: `JWT ${access_token}`,
38            Accept: "application/json",
39            "Content-Type": "application/json",
40          },
41          method: "POST",
42          body: {
43            owner: `${owner}`,
44            name: `${name}`,
45            description: `${description}`,
46          }
47        };
48      },
49      invalidatesTags: ["contracts"],
50    }),

```

Figure 37: cache declaration/invalidation in action

DEVELOPMENT FLOW IN PRACTICE

Development of new features usually follows a pretty concrete consistent process, that takes a lot of the guess work out of “what comes next.” This can really help onboard future contributors because the process for shipping features from start to finish is really well defined. The steps are as follows:

1. Open the project in VSCode
2. Create a new feature-branch off of the develop branch
3. Open terminal and activate virtual environment inside of the project: pipenv shell
4. Run: `python manage.py startapp <appname>`
5. Register the new app in our base settings.py, e.g.: `'contracts.apps.ContractsConfig'`
6. Add the api endpoints in our base *project* level urls.py, including the *app* level urls.py
7. Create new model from wanted data model attributes for that feature in models.py
8. Create serializers for each model in serializers.py
9. Create CRUD or other views in views.py
10. Register the endpoints for the API layer in urls.py, calling upon the created views
11. Create new RTK/RTK-Q slice/api files for that specific feature
12. Register the RTK slice and RTK-Q api inside of our persisted redux store.js
13. Define state changes to the data in the slice if needed
14. Define http request queries in the featureAPI.js file and declare and export autogenerated hooks
15. Create component, call hooks to make requests, and handle the common states associated with them
16. Style the component as needed inside of its respective scss module file, or inline for smaller things

Here we have a concrete 16-step process that will get you from no feature to full feature in no time at all. On the backend, much of the work will be in defining the correct data model and applying the intended app functionality inside of our views. On the frontend, it will be spent defining the api calls, and creating the components containing the logic necessary to complete them.

REFERENCES, TUTORIALS, ETC.

- <https://www.taniarascia.com/react-architecture-directory-structure/>
- <https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow>
- Bashir, Imran. Mastering Blockchain: a deep dive into distributed ledgers, consensus protocols, smart contracts, DApps, cryptocurrencies, Ethereum, and more. 3rd Edition, Aug 2020
- <https://github.com/Bearle/django-web3-auth>
- <https://www.toptal.com/ethereum/one-click-login-flows-a-metamask-tutorial>
- <https://www.youtube.com/watch?v=M576WGiDBdQ&t=16602s>