Contents

[# Abstract 1](#_Toc89478041)

[## Frontend 2](#_Toc89478042)

[## Blockchain 2](#_Toc89478043)

[### History 2](#_Toc89478044)

[### Tools 4](#_Toc89478045)

[# Design Patterns 4](#_Toc89478046)

[### REST MVT pattern: Django Rest Framework 4](#_Toc89478047)

[### Finite State Machine pattern: Redux 4](#_Toc89478048)

[### Persistence Pattern: Redux-Persist 5](#_Toc89478049)

[### Factory Pattern: Redux Toolkit-Query 5](#_Toc89478050)

[# Development Flow In Practice 6](#_Toc89478051)

[# Problems 6](#_Toc89478052)

[# Unknowns 6](#_Toc89478053)

[# What's possible? 6](#_Toc89478054)

[# Future Iterations 7](#_Toc89478055)

[Bibliography 7](#_Toc89478056)

# Abstract

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

# 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 necssary. 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 enviroments like NodeJS.

- <show example screenshots of development directory, workflow, etc.>

- Gitflow Workflow [2]

- 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 of. It also allows for hotfix branches which can test bugs found in production to make sure they're patched before merging into main.

- <show diagram of the flow over time>

- <show terminal screen grab of this process>

## Backend

- Django Rest Framework, PostgreSQL, Ethereum

- For the backend, I chose Django Rest Framework and PostgreSQL. Django is a very robust, full featured framework. It is well documented, there is a lot of built-in functionality and support for common use cases. It's also opinionated enough to allow for total control and customization. As such, Django and in this case the REST implementation, Django Rest Framework, allow for rapidly developing minimum viable products, prototypes, testing out features. The structure breaks features into seperate apps within the project and can even hold multiple sites within the project. This aligns really well with an agile iterative development workflow. New sprint, new app started within the project, new feature branch for it, and suddenly the code is very modular and extensible without stepping on the toes of other parts of the codebase, or even entire sites that other teams may we working on.

- API documentation: swagger/redoc

- built in admin panel for viewing whats happening without needing to query the database manually

## Frontend

- NodeJS, NextJS, useDapp, Redux, RTK, Redux-Persist, RTK-Query, ethers, React-Bootstrap, Scss

- NextJS: image optimization, SSR if wanted, faster, offers more long term extensibility

- react-bootstrap: for quick prototyping, react optimized components

- scss modules: separation of concerns and modularity.

## Blockchain

- Ethereum, Solidity, useDapp, ethersJS, quicknode

### History

- Blockchains, the technology underlying the entire Web3 movement,is no accident. It's the culmintation 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 distrubte 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 transfering 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 oragnizations. 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 times 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 - the real world. amd 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

# # 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.

Text

Description automatically generated

Figure : Smart Contract Model

Text

Description automatically generated

Figure : example post request utilizing models 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

## 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 through the use of \*\*slices\*\*. createSlice function generates all the boilerplate needed to handle the structure of the data, and also 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 ti 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 in 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 simple add a PURGE function to our data mutation on setLogout.

This is not without its cons. There is additional overhead for maining 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 apporach 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.

Text

Description automatically generated

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

Text

Description automatically generated

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

Text

Description automatically generated

Figure : RTK-Q variables being used in conditional rendering of react component

### 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.

Text

Description automatically generated

Figure : 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. Create new model from wanted data model attributes for that feature in models.py
7. Create serializers for each model in serializers.py
8. Create CRUD or other views in views.py
9. Register the endpoints for the API layer in urls.py, calling upon the created views
10. Create new RTK/RTK-Q slice/api files for that specific feature
11. Register the RTK slice and RTK-Q api inside of our persisted redux store.js
12. Define state changes to the data in the slice if needed
13. Define http request queries in the featureAPI.js file and declare and export autogenerated hooks
14. Create component, call hooks to make requests, and handle the common states associated with them
15. Style the component as needed inside of its respective scss module file, or inline for smaller things

Here we have a concrete 15-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.

# Problems

# Unknowns

- centralized vs distributed governance, pros and cons of each.

- centralized vs distributed ownership, pros and cons of each

- misaligned incentive mechanisms, profit max vs integrity max

# What's possible?

- Let's say this project works. We have two parties, who know absolutely nothing about merkle trees or elliptic curve cryptography or programming in Solidity, and yet they're able to sign up for our service, agree to a contract.

- The question becomes, well how many interactions in our society are actually driven by contracts in some form at their core. Everything? Is the answer everything? Insurance companies essentially distribute mass contracts with terms of their coverage that dictate how payouts are made. This is not some small industry or low stake endeavour, and it happens at a massive scale and touches everyone and everything. I'd argue that system is a pretty obvious use case for disruption. You have terms of agreement between a transfer of value, you have real world events that trigger clauses for payouts, and you need some way to bridge that physical event to a digital one to trigger the event. How is that any different from our athletes sports contract with his club? Is there anything as complex and variable as the terms of insurance and its trigger events? If we can solve that, I think we can solve anything.

- Well why would you want to do this? I don't know, why did we start driving cars when we had horses, why did we start flying planes when we had trains. It's just one more efficiency in an ever evolving species. The more problems we solve with this, the more perhaps we realize that this technology is not some toy or scam, but capable of vastly improving transparency, and ensuring people get what they're owed. That people are in good hands, and not being exploited. We want to make a better society. Right? Aren't we all sick of hearing about injustices? if we can start improving lives and solving problems, we can accelerate this movement into Web3.

- We're approaching a tipping point. The Web3 governance model renders all web2 and prior companies obsolete. For those who say, well that's some cypherpunk utopian unrealistic future, I'd argue it's actually the most capitalistic democratic advancement in recent memory. Let's say you were to go to the grocery store, and you wanted to buy soap. Soap A is made by a traditional public company, whose shareholders are mostly endowments, private equity firms, hedge funds, and executives. They maximize for profit, and they distribute those profits to these already rich shareholders. Soap B is made by a DAO company, they also make profits, except this time, the token governance has built in incentive mechanisms that outweigh anything Soap A company can provide. They give you tokens just for buying the soap, they send you a cut of the excess profits, their employees are also compensated for contributing with tokens and profit shares. All things equal, why would anyone buy Soap A? Why would anyone work for Soap A? They'll work for and use the company that provides them the most value, and the intrinsic nature of web3 companies, provides more value by default.

- Competition is no longer a zero sum game, and thus competition is not harmful anymore. We are free to produce and produce because consumption is rewarded. We're just constantly turning this wheel of value, except now it's distributed more evenly. Maybe founders and and major investors still make a lot of money, but because integrity of the system is the core incentive mechanism, they're not the only ones making that money. I read something during the recent John Deere strike where there was some insane statistic mentioned. The company could pay each of its 20,000 on strike workers a lump sum of $250,000, and they would still have $1 billion in profits left over from this year alone. To me, it's just insane, that instead of doing that, they just hoard it all. How many families could that help, how much more would those individuals drive the economy through consumption using their sudden windfall of discretionary income? How much better would society be for it? What is every company did that? This winner take all mentality is not sustainable. It is a perversion of nature, we should give what we can, and take what we need. Not a few people take everything, and everyone else gives everything. There's no balance or harmony. It's no wonder society is crumbling. We're doing it wrong.

# Future Iterations

# Bibliography

[1] https://www.taniarascia.com/react-architecture-directory-structure/

[2] https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

[3] Bashir, Imran. Mastering Blockchain: a deep dive into distrubted 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://usedapp.readthedocs.io/en/latest/index.htmlw