CRPL: Report

Harrison Beau Barker

Supervised by Dr Dan McQuillan

https://github.com/MrHarrisonBarker/crpl

Hbark002, 33575210

Abstract

CRPL (**C**opy**R**ight on a **P**ublic **L**edger) is a Ethereum blockchain backed system for registering and maintaining copyrights of digital works, this system is accompanied by a web app allowing users to easily interact with the system

Contents

1 Keywords

Copyright

Blockchain

Ethereum

Smart Contract

EVM

2 Introduction

The aim of this project is to represent a version of ***copyright*** for protection of intellectual property on a ***blockchain*** backed by a public ledger of transactions. My initial impetus for this project was a book I read in 2018 called *"Blockchain Revolution: How the Technology Behind Bitcoin Is Changing Money, Business, and the World"*[20], before this point I knew little of the applications for blockchain technologies attributing them only to a new form of digital currency allowing peer to peer transactions of wealth.

However, this book introduced the concept of ***smart contracts*** and the possibilities now small immutable programs could be saved and run on a blockchain. The most relevant change smart contracts brought was progressive state into an infamously immutable technology, by leveraging an unmodifiable chain of transactions state became not just a current record (like most traditional systems) but a historical account of all previous states with a clear a definable list of transformations precisely timestamped.

This new knowledge of ***blockchains*** came to fruition when it came to selecting my final project but to what problem should I help to provide a solution to leveraging this technology. A combination of recent interest in the crypto-sphere mainly coming from NFTs, continued displays of a ***copyright*** system not fit for purpose[8] and the book I had read 3 years earlier proposing that ***blockchain*** and ***smart contracts*** were an extremely viable solution to problems intersecting law and social structures.

2.1 Unfit for purpose

So why is the current ***copyright*** system not fit for purpose? I've broken it into three interlinking factors; complexity, jurisdiction dependence and lack of digital computerised systems. starting with complexity, it is often hard as a creator to know if your work is protected or if the protection is enough? This gives massive power to publishers, managers and companies who are willing to exploit this fact by blinding a creative with a large contract covered in legal jargon. Should an artist also have to be a lawyer?

Jurisdiction dependence can be looked at as a point of complexity and inefficiency in the system, yes there's international ***copyright*** law in the form of the *"Berne Convention"*[1] and later *"WIPO Copyright Treaty"*[9] which informs most of our modern ***copyright*** law. However, these treaties only state minimum requirements and standards to follow but do not control the internal ***copyright*** law of a sovereign nation which will always take precedence.

This idea of global ***copyright*** links nicely into my last factor which is centred around the digital and ever more interconnected world we're living in. Even though in 1996 the **WCT**[9] was introduced to address issues caused by the emergence of information technology, however the world and more specifically the internet has changed an unimaginable amount since 1996 but ***copyright*** is effectively unchanged including the systems to register and view registered ***copyrights*** which are closed off in obscurity.

2.2 What a solution needs

I've defined what I believe as to be four requirements of a solution to this problem and how I've addressed these requirements.

Global Both the system needed to be globally accessible, available and consistent across all jurisdictions to minimise complexity and maximise protection for an interconnected world. I've implemented this by using the ***Ethereum*** network which is made up of thousands of nodes across the world.

Open Openness is essential to instil trust in a completely independent and alternative solution compared with government institutions that have implied trust and guaranty. I've implemented this by writing and using open-source code including ***Ethereum*** which is open-source and backed by a public ledger.

Robust The current written laws and contracts maybe complex but are strongly defined with a commonly accepted interpretation, this will have to be true in this solution. I've implemented this by using an immutable ***smart contract*** for copyright representation and logic.

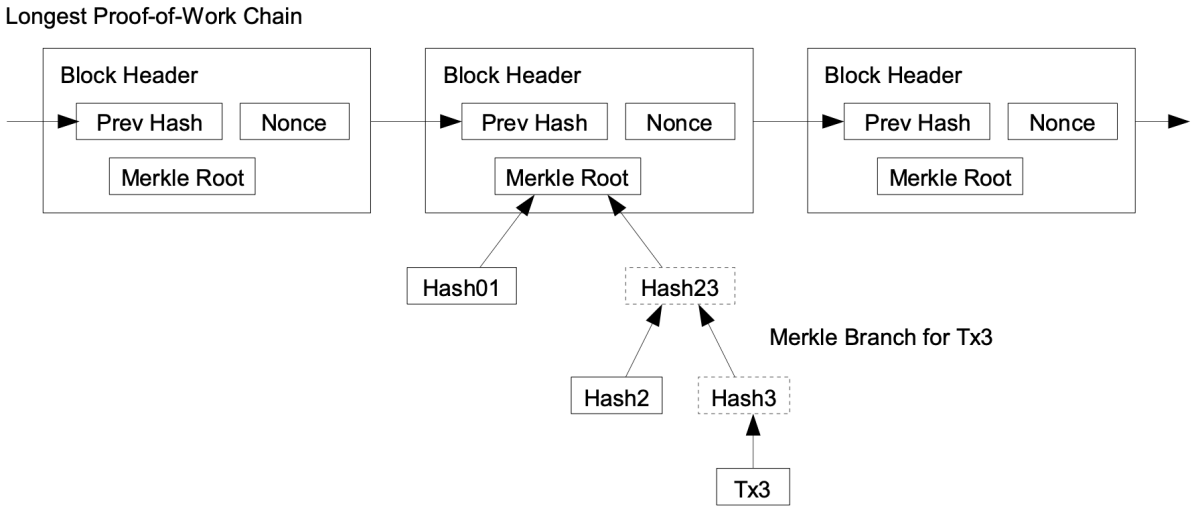
Simple This is the most important requirement as the proposed problem is centred around current ***copyright*** complexity so any solution will need to be accessible to every possible user. I've implemented this by using clearly defined selectable ***copyright*** protections when registering a work.

3 Research

3.1 Blockchain

The theory of a ***blockchain*** is not new [18] but the implementation and hype around ***blockchains*** is new and extremely popular in the current day thanks to Bitcoin designed by Satoshi Nakamoto based on their white paper [14]. The accepted benefits of a ***blockchain*** are decentralisation, distribution, and immutability.For a ***blockchain*** to exist and be useful it needs nodes to maintain the data structure authenticate all transactions and hash the current block in preparation for the next in the chain.

Figure 1: Blockchain structure with transactions saved in Merkle trees[2]



3.1.1 Technical proficiencies / limitations

Proficiencies

Security Is the core of ***blockchain*** as the cryptographic hashing strategy is what makes the data structure a chain.

Transparency Is not intrinsic to ***blockchains*** that is down to the implementation (public or private), however the underlying concept facilitates and promotes this open and transparent behaviour simply by the fact that nodes within a ***blockchain*** network need a copy of all transactions/blocks.

Decentralisation Is definitely the most popular benefit of ***blockchains*** and it's not a technical benefit just like transparency decentralisation is a social debate that is made possible by technical decisions of the ***blockchain*** architecture, centred around the control and ownership of peoples data

> "It avoids concentrations of power that could let a single person or organization take control."[11]

Limitations

Blockchain technology faces two major limitations, the largest chains in the world (Bitcoin and ***Ethereum***) are quickly growing in size which has pointed out scaleability problems caused by transaction speed and cost per transaction which are currently slow and expensive respectively compared to traditional systems (cost

experiments have borne this out in the aptly named paper "Comparing blockchain and cloud services for business process execution"[15] and show that business logic on ***blockchains*** are 2 orders of magnitude more expensive than current cloud services).

Blockchains rely on ***consensus protocols*** which is a mathematical formula to determine consensus of the network, essentially enough of the nodes have to agree the current state of the ***blockchain*** whenever it's modified. Currently all major chains including Bitcoin and ***Ethereum*** use a protocol called proof of work[7] which uses the total amount of compute a node has produced as the comparable proof that can be verified by many other nodes easily. This has worked so far for these major chains however ballooning energy consumption (which can be seen in realtime at the Cambridge Bitcoin Electricity Consumption Index), high transaction costs, and inadequate transaction processing bandwidth is forcing them to change and look for solutions.

Ethereum have decided to completely reinvent their consensus protocol with a fundamentally different protocol called proof of stake which promises reduced energy consumption, cheaper, and faster transactions[6]. Although there have been other proposed solutions including: improving the proof of stake system by parallelising it across nodes[17] or reducing the block size for a increase transaction capacity by reducing the amount of work needed per block with a security trade off to balance[12].

3.1.2 Use of blockchain for IP protection

I am not the first person to have thought of representing and enforcing ***copyright*** protection on a distributed ***blockchain***, there has actually been a lot of discussion around the topic showing the benefits of using this technology for ***copyright*** such as transparency;

> "Blockchain may substantially increase visibility and availability of information about copyright ownership." [16]

the power of ***smart contracts*** and utilising a networks cryptocurrency

> "Smart contracts will allow automatic and instantaneous payments to designated parties, and expiration of a license after a certain amount of time." [16]

and simplification through the globalisation of ***copyright*** law. majority opinion says that introducing ***smart contracts*** to ***blockchain*** technology is extremely powerful particularly within ***copyright*** law to

> "reliably automate a large volume of 'dumb transactions'" [10]

which will greatly reduce friction by removing unnecessary work and removing expertise needed currently in the field to be properly protected.

3.1.3 Use of blockchain for legal purposes

Bringing ***copyright*** to ***blockchain*** is apart of a larger conversation about the compatibility of any legal or even governmental workflows to be either represented or completely reinvented using ***blockchain*** technology, and it does look like in many cases these types of problems can leverage ***blockchain*** and possibly even thrive.

The legal industry is notoriously complicated requiring a great level of knowledge and expertise in the field to make sense of anything or more importantly get

anything done. Logically laws make an obvious starting point for computerisation because computers are defined systems of ridged laws however this didn't make sense during the first wave of ***blockchains*** which almost entirely focused its efforts towards currency. The introduction of ***smart contracts*** has opened up the possibilities for automated law processing massively reducing boilerplate and bulky legal work which is largely trivial but time consuming.

> "So-called 'smart contracts' built on blockchain technologies may prove to be the most important example yet of "self-executing, customised rules"." [13]

For governing ***blockchains*** ledger and transparency will take centre stage as essentially all governments are big collections of "things", assets, people and information. Not only are they collections but historical records, thankfully ***blockchains*** are immutable, timestamped, secure and built for openness. Because ***blockchains*** are just general purpose data structures there're uses are broad;

> "keeping an overview of the authorities provided in a public organization and the ability to change the authority only if there is agreement among nodes which are classified as being higher ranked in the hierarchy." [22]

3.2 The market

The vast majority of ***blockchain*** applications in the current day are financial for obvious reasons with the market capitalisation of cryptocurrencies being the universal metric for the ***blockchain*** markets size [21], however the health care industry has been aggressively investigating and implementing ***blockchain*** technology particularly in the secure distribution of medical data and health records [19], supply chain technology has seen some innovation tracking goods as they pass through a chain.

4 Design

4.1 Functional Requirements

This is a list of key required features needed for this project to be successful.

Figure 2: Functional requirements

Feature	Function	Priority
Copyright smart contract	Immutable code on a public ledger "blockchain" for the purpose of establishing ownership or the copyright to a piece of work.	CORE
Multi-party distribution	The ability to establish a complex ownership structure of multiple individuals/groups, all these owners will have equal ownership of the copyrighted work.	CORE
Ownership transfer	The ability to change the ownership of a copyright from one complex structure to another with consent of all current owner(s).	CORE
Work verification	Verification of a work to establish its originality with a reasonable accuracy for the platform, no duplicate works will be allowed to be registered.	CORE
Dispute filing	Allow any user to dispute a copyright with sufficient evidence and provide an option for resolving these disputes by the owner(s).	CORE
Digital signing	Digitally sign a registered work with unique data that identifies it as original copyright registered, much like the traditional copyright registration symbol.	CORE
Decentralised Work CDN & proxy	Store registered works on a decentralised technology providing a free and open option for creators.	EXTRA
Websocket updates	Real-time updates for the front-end UI to provide a better end-user experience.	EXTRA

4.2 Smart contract

4.2.1 Inspiration

To design a smart contract without prior experience I decided to look at an existing contract and because I knew my contract was going to exhibit similar functionality and principles as **NFTs** I started with the EIP[i] for non-fungible tokens EIP-721 .

This describes a standard interface for all external methods and events **NFT** contracts need to implement, most are self-explanatory like balanceOf, ownerOf, transferFrom and the Transfer event. Then there's a methods concerning "approval" which is ***Ethereum*** speak for access control, essentially what addresses are allowed to transact with a token.

[i] "Ethereum Improvement Proposals (EIPs) describe standards for the Ethereum platform, including core protocol specifications, client APIs, and contract standards."[4]

Interfaces are useful for understanding how the contract interacts with the outside world but not the internal working. So I found an implementation here written by OpenZeppelin to gain an understanding of the internal operation.

Figure 3: ERC721.sol data mappings

```
1  // Mapping from token ID to owner
2  mapping(uint256 => address) private _owners;
3
4  // Mapping owner address to token count
5  mapping(address => uint256) private _balances;
6
7  // Mapping from token ID to approved address
8  mapping(uint256 => address) private _tokenApprovals;
9
10 // Mapping from owner to operator approvals
11 mapping(address => mapping(address => bool)) private _operatorApprovals
       ;
```

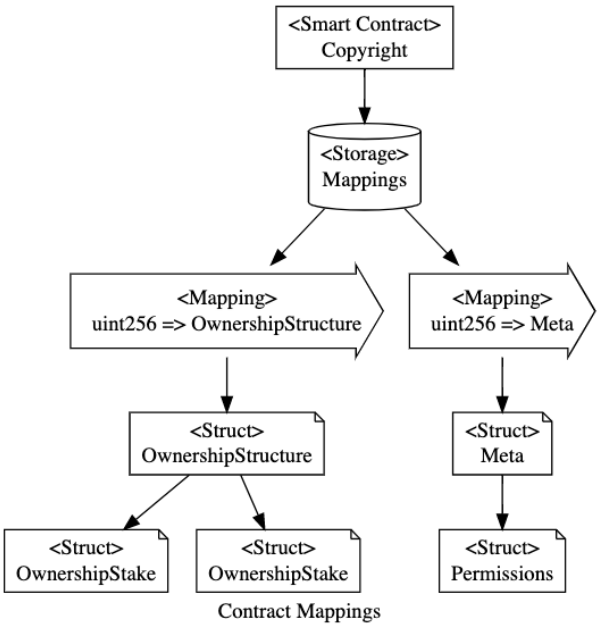
This snippet is taken from the OpenZeppelin contract and is essentially how an **NFT** "works". a series of mappings that are saved in storage[ii] and are hash-maps from one type to another. *_owners* points to an address based on the input token id (uint256) all methods in the contract simply modify these mappings for example when you register a token your wallet address is saved in the map entry for the next id.

4.2.2 Ownership structure

This architecture was used as foundations for my new contract, however there is one major requirement of my system not supported by the EIP-721 standard which is multi party ownership of a token. This alone wasn't going to represent copyright as works can quite often involve multiple people collaborating, the book I mentioned at the beginning of this report[20] has two authors, a system not representing the work and effort of all involved is not acceptable.

[ii] storage is an area of the ***EVM*** that every smart contract has access to for storing state variables that need to persistent. memory and stack are also available

Figure 4: Structured Ownership essential mappings



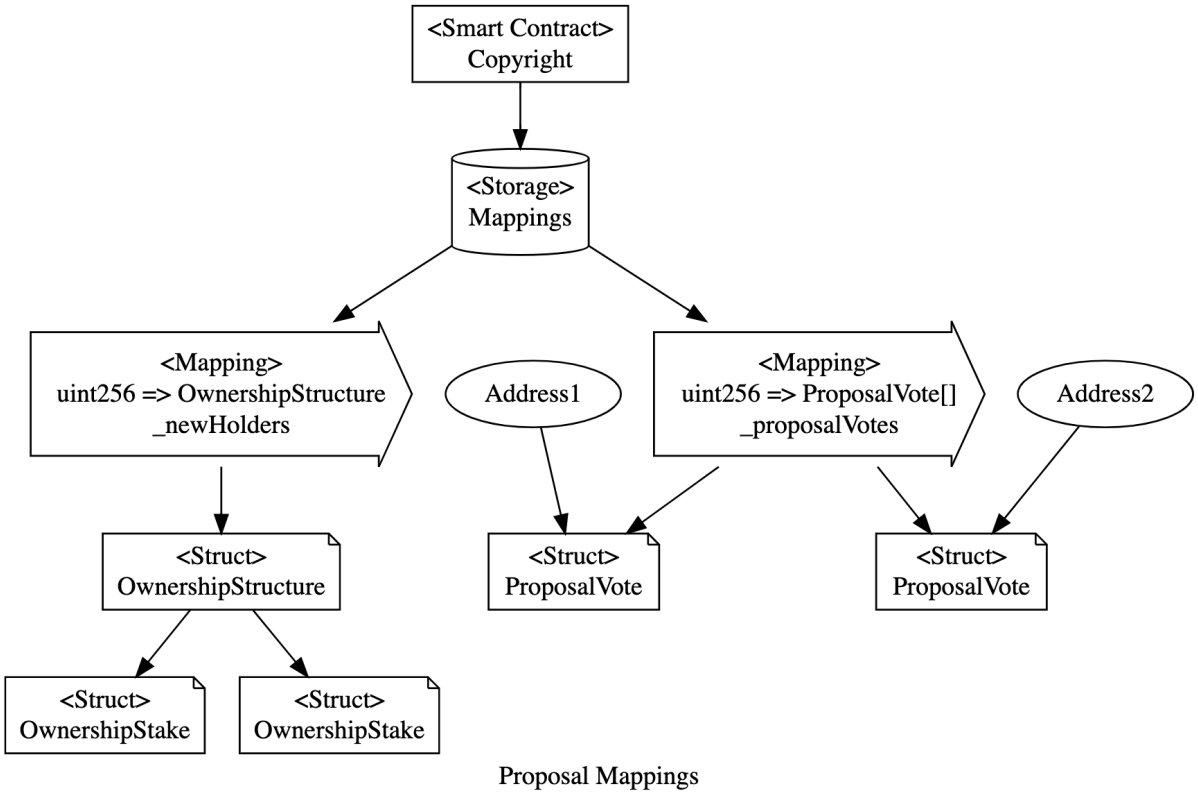
To solve this problem I've redesigned from the ground up how ownership is defined in the contract, instead of mapping token ids to one address the contract now maps to an **OwnershipStructure** that contains a list of owners along with a number of shares that specific address holds in the token. This design clearly borrows from a limited company share structure allowing a complex ownership of multiple individuals or groups with implied variance in ownership[iii].

4.2.3 Shareholder consensus

Allowing multiple wallets to own a token now introduces a new problem for contract design, when a change is made everyone has to agree just check if you're an owner is not enough anymore, allowing everyone with a stake to make changes without consulting all other owners is a point of exploitation in need of solution.

[iii] Although the number of shares an address owns makes no immediate difference in the current implementation of this contract as this was outside of the desired complexity scope.

Figure 5: Structured Ownership proposal mappings



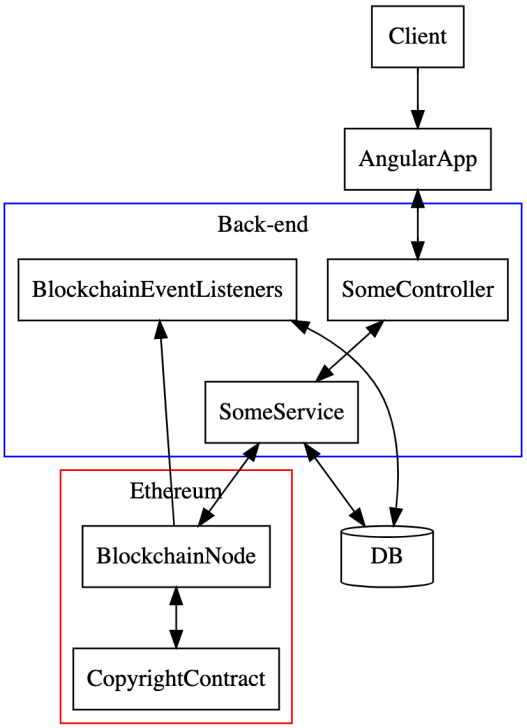
I've designed this solution for the shareholder consensus problem, now instead of making direct changes to a copyright (above is an ownership restructure) a user proposes change to a copyright which is then voted on by all owners until a unanimous vote is reached then the change is made.

4.2.4 Protections

To simplify and give users customisability of protection I've designed a protections system similar to permission in many computer systems, the list of available protections was built from existing legal protections provided by ***copyright*** law [3] including: Adaptation, Performance, Reproduction and Distribution.

4.3 Back-end

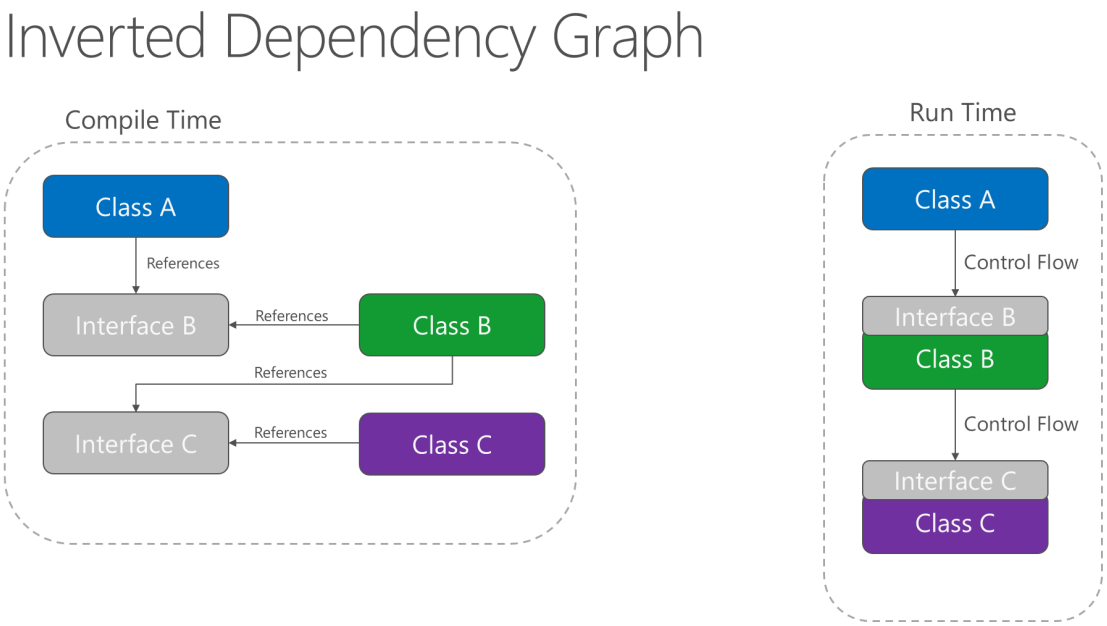
Figure 6: Back-end abstract operation



4.3.1 Dependancy injection

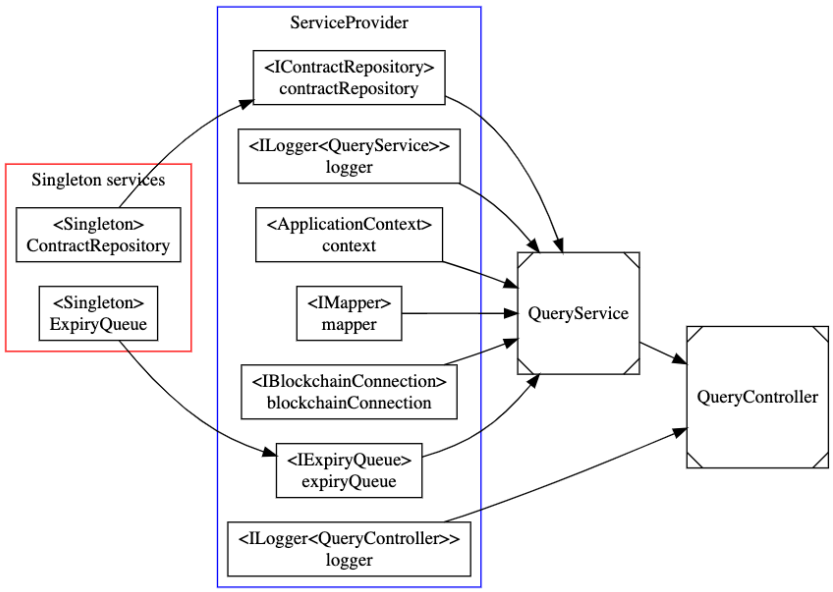
Dependancy injection is a supported and encouraged design pattern within **.NET** and **ASP.NET** which allows building loosely coupled applications by separating implementation and design, depending on a softwares design opposed to its technical implementation results in more resilient and modular code.

Figure 7: DI graph taken from docs.microsoft



Above is an inversion of control and dependancy injection example, as you can see each class is depending on an interface of the desired class not the actual code implementation hence loosely coupled.

Figure 8: Example dependancy graph for the query controller and service

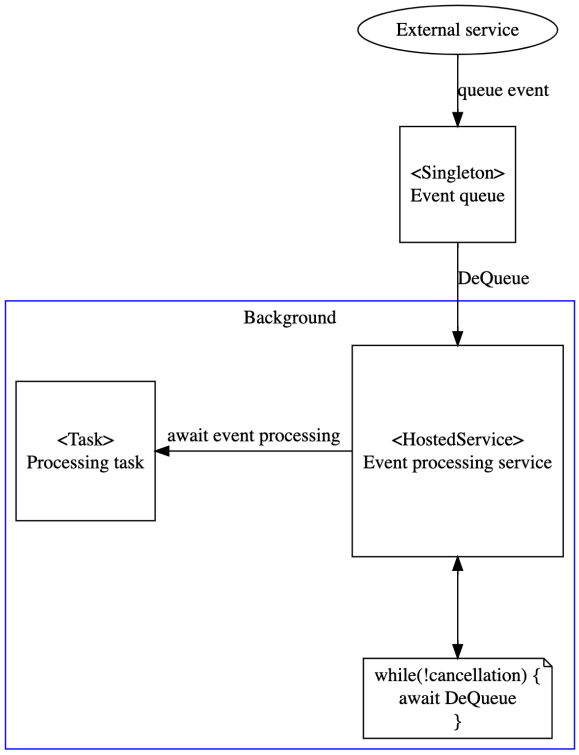


This is real example of dependancy injection drawn from the query controller and service injections. It shows both don't depend on any implemented classes just the interfaces which describe how you can interact with an implemented version of that class. At runtime each interface injected will be populated from the service collection with a real class that was registered on startup.

4.3.2 Background services

Background services was designed based on previous work I wrote for OpenEvent so instead of using new software like Hangfire which is more feature rich with a strong following I decided for the scale of this project and the existing pattern I had developed and knew intimately a year ago would be a better fit.

Figure 9: Queued background service pattern



Background processing pattern

This is the basic pattern describing how my background services work, it's essentially a queue and processing service. 'Work' is queued while a processing service running in its own thread dequeues work and processes, after this work completes the processing service waits for the next item to dequeue.

This pattern can be quickly implemented and tailored to specific background work, it's easily scalable with the number of threads configurable in the processing service.

4.3.3 Event processing

Because transactions on the ***blockchain*** can take any amount of time to be verified, placed into a block and for that block to be placed onto the chain. Blocks on ***Ethereum*** are processed around every 13 seconds but you're not guaranteed to be placed into the next block, this largely depends on the amount of **gas** you're willing to spend and the volume of current transactions.

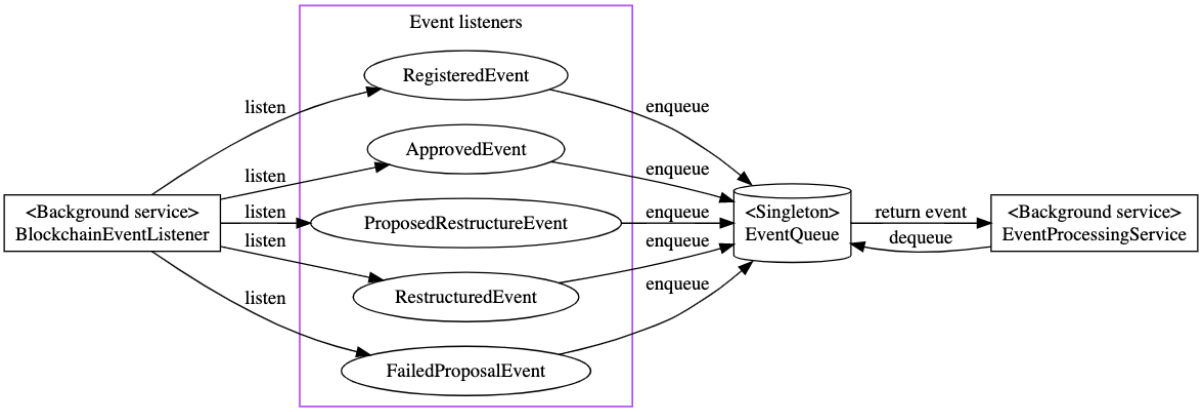
All this means that my system must be able to send a transaction then wait an indeterminate amount of time for a response meaning I can't force the user to wait on that transaction until complete. Thankfully ***Ethereum*** has a solution for this called **Events**, I've specified a number of these events in my contract (see below) which the back-end "listens" for by processing the information in each block.

Figure 10: IStructuredOwnership events

```
/// @dev Emits when a new copyright is registered
event Registered(uint256 indexed rightId, OwnershipStake[] to);

/// @dev Emits when a copyright has been restructured and bound
event Restructured(uint256 indexed rightId, RestructureProposal
    proposal);

/// @dev Emits when a restructure is proposed
event ProposedRestructure(uint256 indexed rightId, RestructureProposal
    proposal);

/// @dev Emits when a restructure vote fails
event FailedProposal(uint256 indexed rightId);
```

When an event is found it gets added to a processing queue then a processing service dequeues each event and processes based on the type of event.

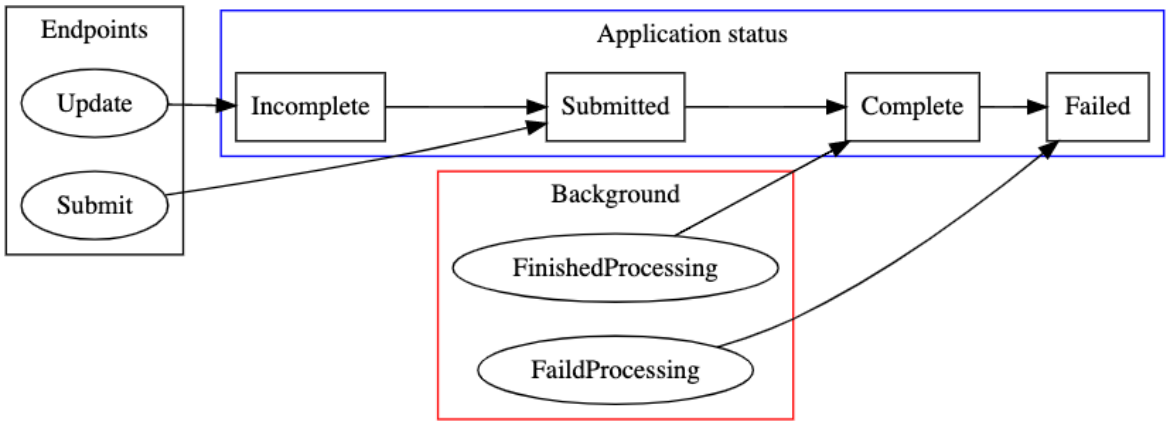
Figure 11: Blockchain event listeners and processing



4.3.4 Applications framework

Handling forms and applications is awkward and full of edge cases, the code for handing these applications (eg: copyright registration) can become large and convoluted especially when your system implements many. For this system five applications are needed: copyright registration, ownership restructure, dispute, wallet transfer and delete account. I decided to design a solution for handling application flow and state as a generic process, this means all applications will follow the same state flow and interaction endpoints *seen below.*

Figure 12: Applications framework state diagram

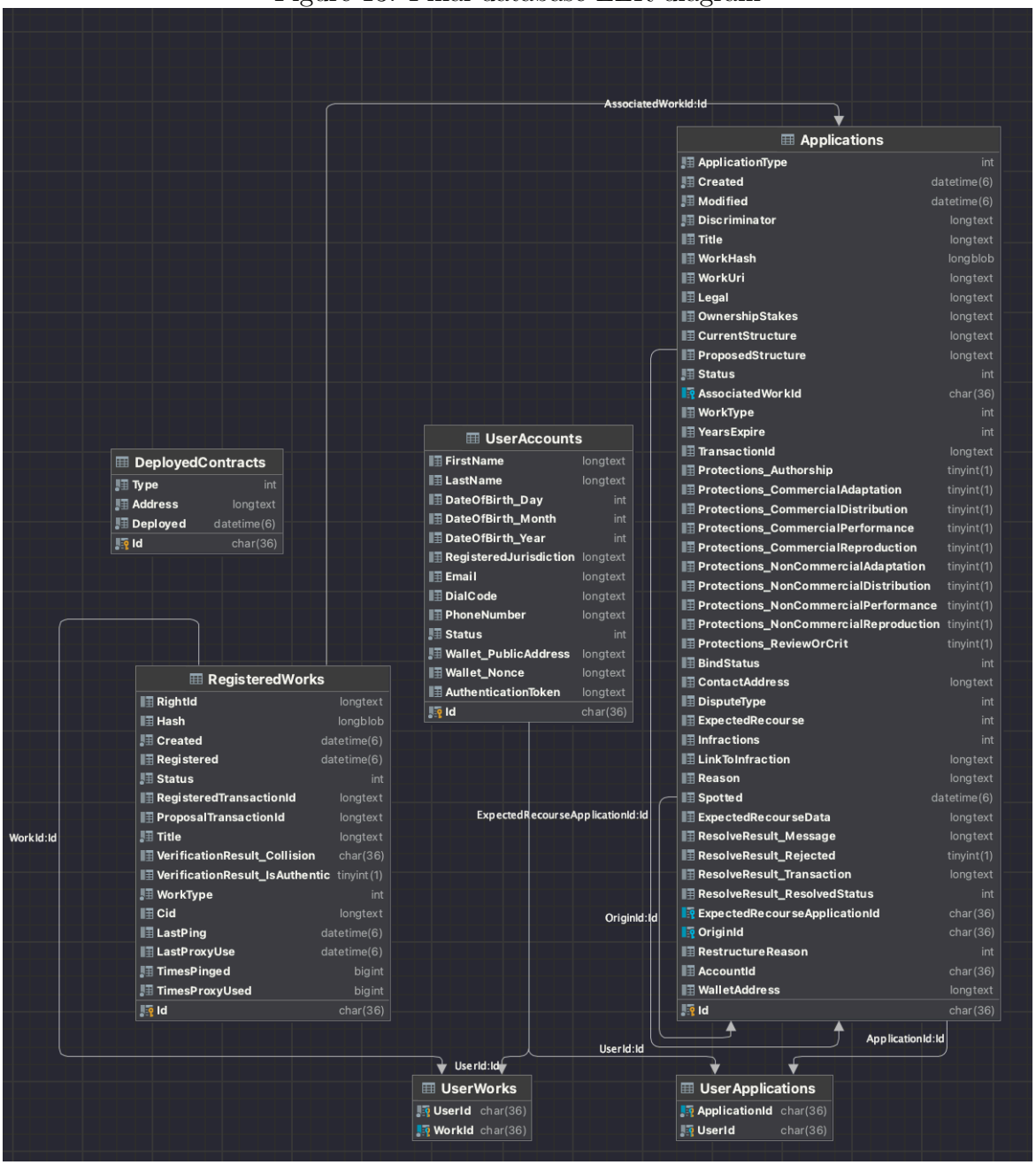


This generification and rigid design flow proved extremely useful, this was because a clear view of state is essential when maintaining parity between systems.

4.4 Database

The database needs to store two types of data: data stored on both the ***blockchain*** and CRPL (eg: public wallet addresses, contract address, and registered works) and data stored solely on the database not mirrored with the chain (eg: applications, user account information and explicit database relationships).

Figure 13: Final database EER diagram



4.4.1 Chain parity

Some data needs to be kept in parity with the ***blockchain*** this introduces a problem because data on the chain can change independently from the system, a user could transact with the ***smart contract*** and register a new copyright or propose a new ownership structure and even bind that new structure destroying any continuity between the database and what's real. This will lead to a terrible user experience but I can't stop it, the chain is open to anyone and is the entire point of this project.

However I can react to change, everything is open and accessible all I have to do is read the ***blockchain*** and update the database accordingly keeping in mind the chain is always the source of truth not the database.

4.4.2 Independent from the chain

I've chosen to keep certain data off the ***blockchain*** only representing in the database. It's possible to store everything on chain completely independent of any database, however this would ballon the size of my ***smart contract*** which are limited to 24KB complied it would also hamper maintainability and future development.

Imagine everything is represented on the ***smart contract*** including dispute handling and applications, the contract is deployed on the chain and now running in the ***EVM*** what happens when I want to add a new type of application or discover my dispute handling is un-ethical or exploitable? I can't change the smart contract it's immutable, the only solution is deploy a new contract and manually migrate all previous ***copyrights***, however this would be expensive, slow and arguably against the spirit of ***blockchain*** technology and the law as I'm effectively changing the underlying representation of a ***copyright*** without consultation or approval.

4.5 Front-end

Visual design of the web application was on lowest priority so a focus on pure usability and function was the goal because of the complex undertaking needed to implement all functionality building effectively two backend systems (***blockchain*** and web API).

I decided to use the Clarity design system and libraries as they have support for Angular with an enterprise/function first focus.

Figure 14: Original dashboard page wireframe

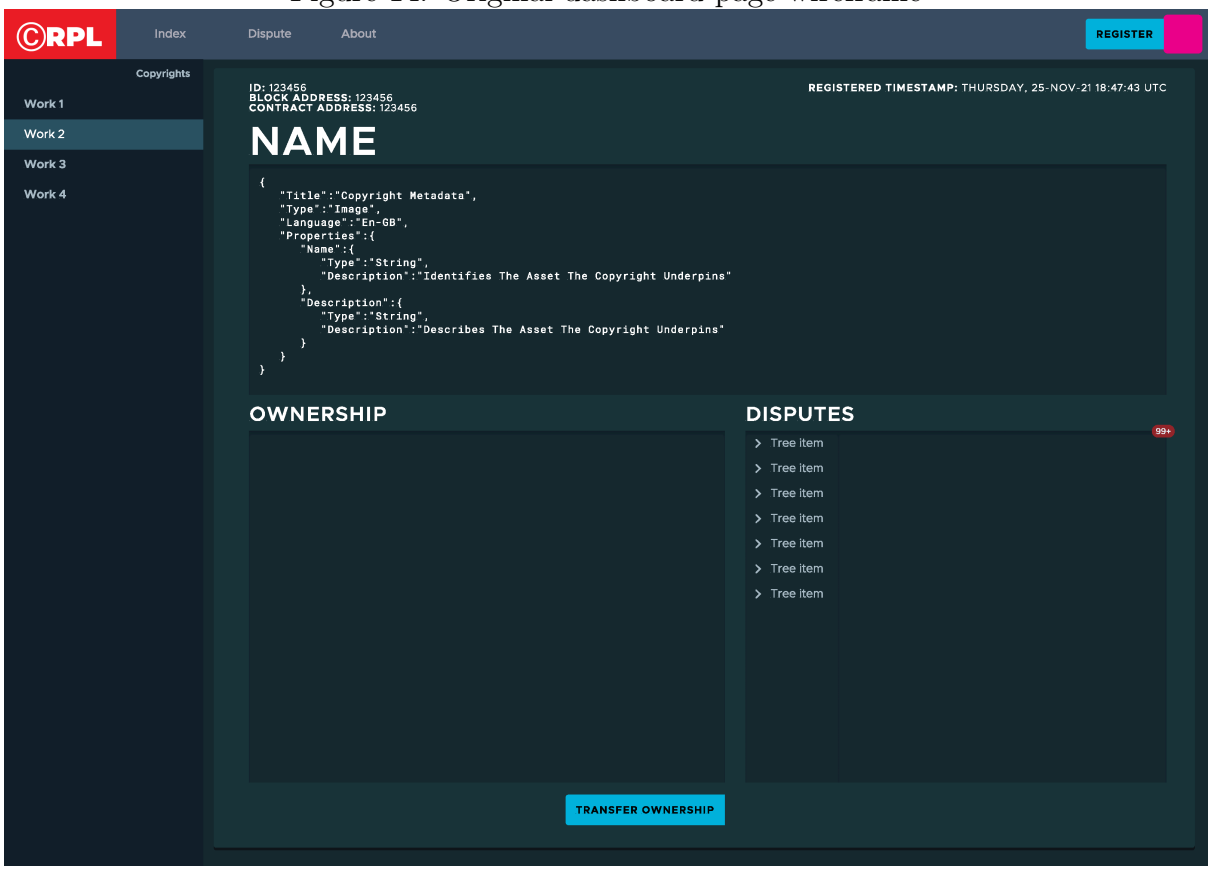


Figure 15: Final dashboard design

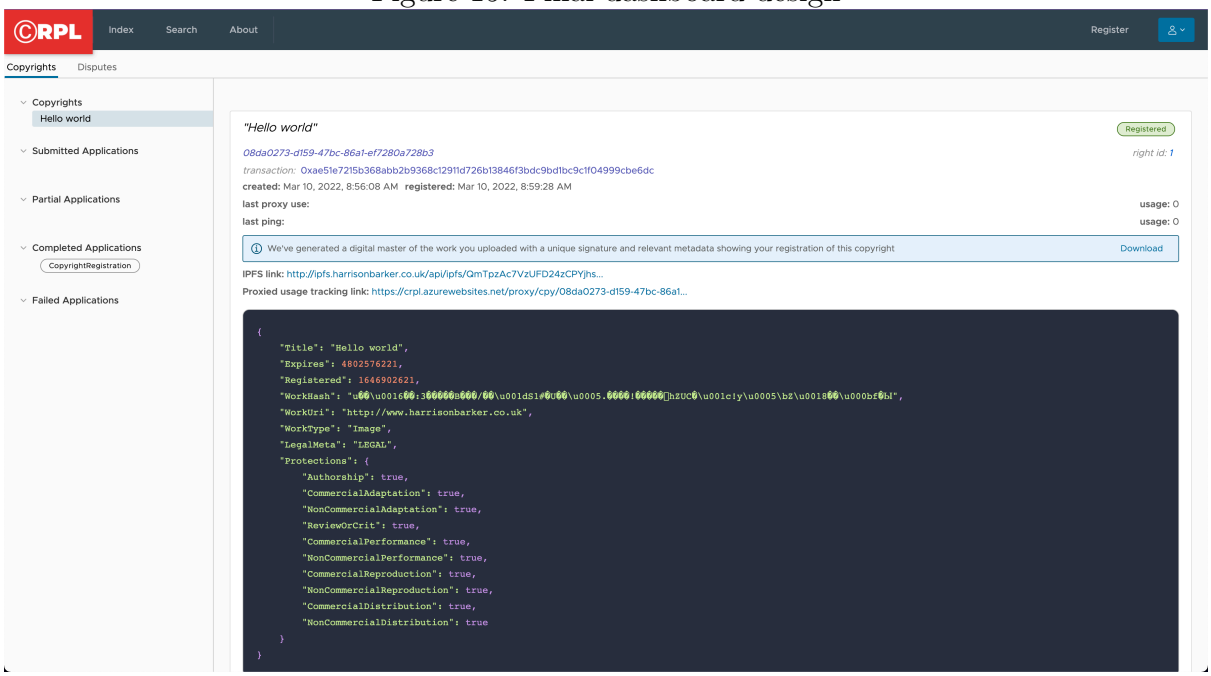


Figure 16: Original register form wireframe

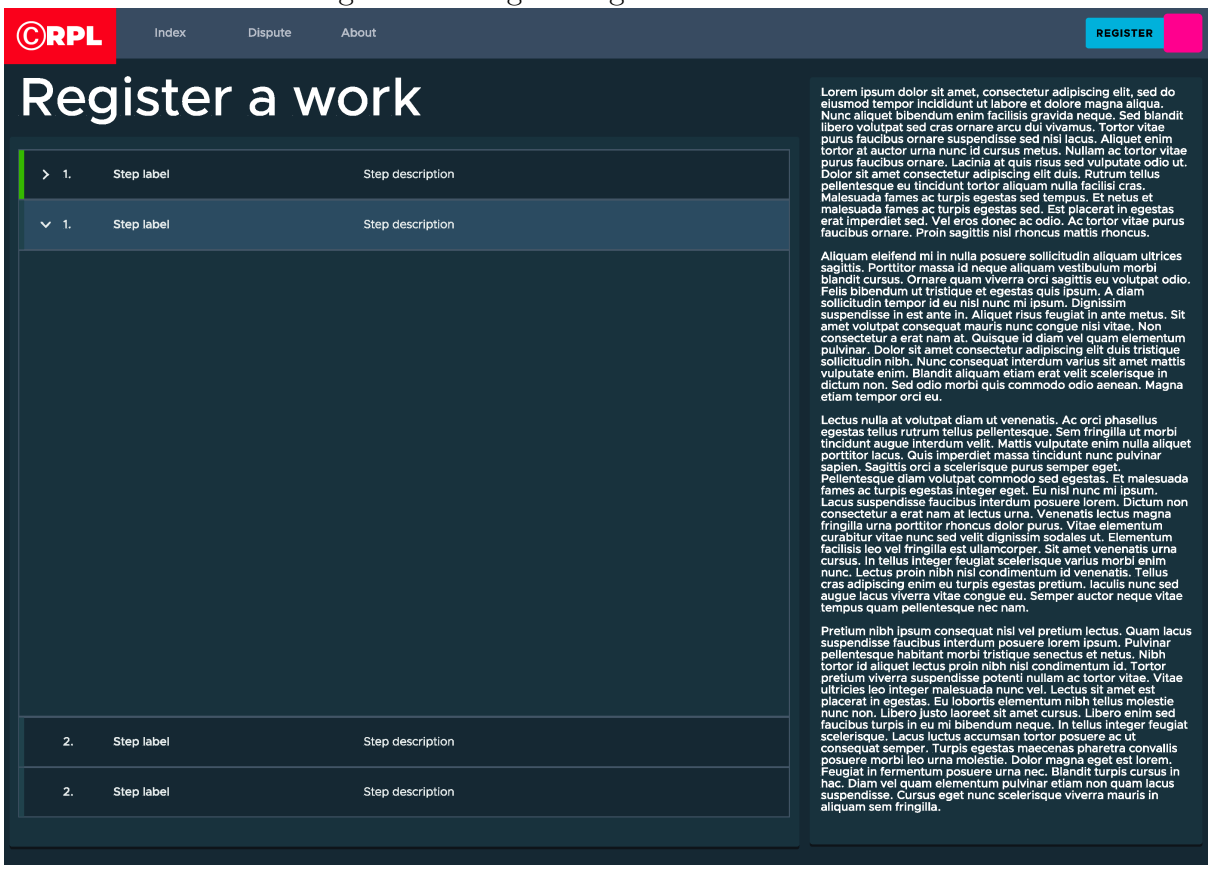
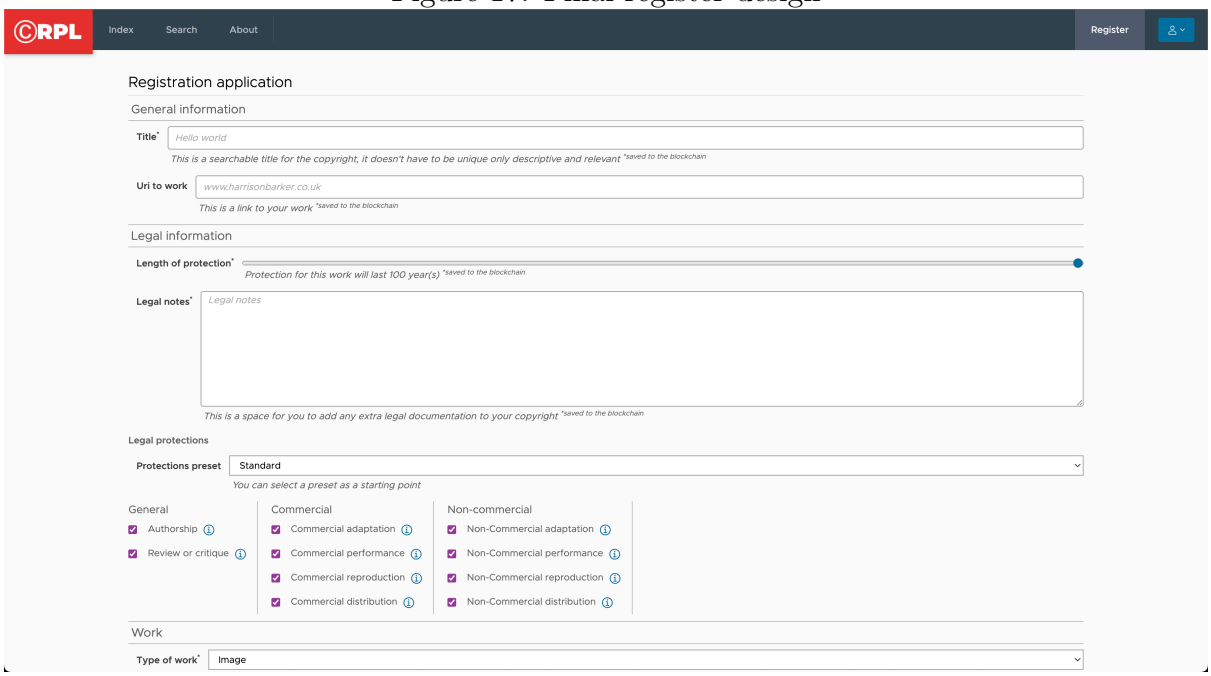


Figure 17: Final register design



4.6 Development process

Development was split into four sprints, two larger sprints two weeks long to start development focusing on major core features essential for any success of the project: ***smart contracts***, contract interaction, registration and restructure applications and user authentication. Followed up with two one week sprints focusing on secondary features and quality of life: disputes, search, synchronisation and account config.

Sprint	Start	End	Length
1	19 January 2022	2 February 2022	2 weeks
2	4 February 2022	18 February 2022	2 weeks
3	20 February 2022	27 February 2022	1 week
4	1 March 2022	8 March 2022	1 week

5 Implementation

5.1 Smart contract

5.1.1 Interface overview

The contract interface is split in two, IStructuredOwnership.sol which handles multi-ownership transactions and events and ICopyright.sol which handles all ***copyright*** transactions and events (similar to the EIP-721 interface).

Figure 18: ICopyright events

```
/// @dev Emits when a new address is approved to a copyright
event Approved(uint256 indexed rightId, address indexed approved);

/// @dev Emits when a new manager has been approved
event ApprovedManager(address indexed owner, address indexed manager,
    bool hasApproval);
```

Figure 19: IStructuredOwnership events

```
/// @dev Emits when a new copyright is registered
event Registered(uint256 indexed rightId, OwnershipStake[] to);

/// @dev Emits when a copyright has been restructured and bound
event Restructured(uint256 indexed rightId, RestructureProposal
    proposal);

/// @dev Emits when a restructure is proposed
event ProposedRestructure(uint256 indexed rightId, RestructureProposal
    proposal);

/// @dev Emits when a restructure vote fails
event FailedProposal(uint256 indexed rightId);
```

First looking at all the events most are straight forward, *Approved* and *ApproveManager* are taken from the EIP-721 standard (ApproveManager == ApprovalForAll) then for *IStructuredOwnership* I've introduced four new events enabling modifiable multi address ownership with consensus.

Figure 20: IStructuredOwnership functions

```
/// @notice The current ownership structure of a copyright
/// @param rightId The copyright id
function OwnershipOf(uint256 rightId) external view returns (
    OwnershipStake[] memory);

/// @notice Proposes a restructure of the ownership share of a
    copyright contract, this change must be bound by all share holders
/// @param rightId The copyright id
/// @param restructured The new ownership shares
/// @param notes Any notes written concerning restructure for public
    record
function ProposeRestructure(uint256 rightId, OwnershipStake[] memory
    restructured) external payable;

/// @notice The current restructure proposal for a copyright
/// @param rightId The copyright id
/// @return A restructure proposal
function Proposal(uint256 rightId) external view returns (
    RestructureProposal memory);

function CurrentVotes(uint256 rightId) external view returns (
    ProposalVote[] memory);

/// @notice Binds a shareholders vote to a restructure
/// @param rightId The copyright id
/// @param accepted If the shareholder accepts the restructure
function BindRestructure(uint256 rightId, bool accepted) external
    payable;
```

Four new and one modified functions enable this functionality, *ProposeRestructure* to propose a change of ownership and *BindRestructure* to vote for proposals. *OwnershipOf* is modified *ownerOf* from EIP-721 which originally returns an address but now returns a complex ownership structure.

5.1.2 Registration

Registration of a ***copyright*** as discussed in the 4.2 uses basic principles of existing contract implementations most importantly the EIP-721 contract, which registers new tokens by assigning a wallet address in a token ids mapping entry stored on the contract. This is simple and secure as the resulting hash of a token id will always point to the same entry therefore address.

Figure 21: Essential copyright mappings from CopyrightBase.sol

```
// rightId -> ownership structures
mapping (uint256 => OwnershipStructure) internal _shareholders;

// rightId -> metadata
mapping(uint256 => Meta) internal _metadata;
```

I've taken this design and modified it for my needs by mapping to a complex ownership structure instead of one address (enabling multi-ownership), I've then

added a new mapping to save metadata (work hash, expiry date, legal protections) for each ***copyright***. These technically could be merged into one mapping along with the four others, mapping from id to a larger more complex struct encompassing all data needed. However I wanted to keep the size of my data structures small with their own defined purpose, this also reduces transaction costs because only the data relevant is being processed.

Figure 22: Register function from CopyrightBase.sol

```
1  function Register(OwnershipStake[] memory to, Meta memory meta) public
       validShareholders(to) {
2
3    uint256 rightId = _copyCount.next();
4
5    // registering copyright across all shareholders
6    for (uint256 i = 0; i < to.length; i++) {
7
8      require(to[i].share > 0, INVALID_SHARE);
9
10     _recordRight(to[i].owner);
11     _shareholders[rightId].stakes.push(to[i]);
12   }
13
14   _metadata[rightId] = meta;
15   _shareholders[rightId].exists = true;
16
17   _approvedAddress[_copyCount.getCurrent()] = msg.sender;
18
19   emit Registered(rightId, to);
20   emit Approved(rightId, msg.sender);
21 }
```

As you can see from the above registration is straightforward: generate the next id, iterate over all shareholders inputted, add each shareholder to the mapping checking each has some shares, add metadata to mapping, approve the sender for this ***copyright*** and emit a registered and approved event to let the system know whats happened.

Lastly I want to focus on line 8, *require* is apart of **Solidity**'s error handling. The ***EVM*** runs all functions transactionally in the programming sense meaning all changes to the persistent data structure are processed and saved after the function has completed successfully. This is extremely useful and greatly simplifies error handling because if we encounter an error no underlying changes to the data have taken place and the transaction simply fails.

Require therefore checks an expression is true and if not an error is thrown with a stated reason the transaction is canceled and no changes to stored data structures are made. These are used throughout my ***smart contract*** to validate input data, I talk more about these in 5.1.4 below. This *require* on line 8 is checking all shareholders have more than zero shares otherwise throw an error with "INVALID_SHARE".

The events emitted from this function are "listened" to and processed in the back-end more information in section 5.2.6.

5.1.3 Ownership restructure

As discussed in ***smart contract*** design I had to build a shareholder consensus function from scratch for making changes to the ***copyright***, therefore changing the ownership structure is split into two functions/steps, first you propose a new structure then each shareholder must vote or *"bind"* the new structure, when all the shareholders have agreed to the new structure mappings are updated to reflect the new ownership.

Figure 23: ProposeRestructure function from CopyrightBase.sol

```
1  function ProposeRestructure(uint256 rightId, OwnershipStake[] memory
       restructured) external override validId(rightId) isExpired(rightId)
       validShareholders(restructured) isShareholderOrApproved(rightId, msg
       .sender) payable {
2
3          for (uint256 i = 0; i < restructured.length; i++) {
4
5              require(restructured[i].share > 0, INVALID_SHARE);
6
7              _newHolders[rightId].stakes.push(restructured[i]);
8              _newHolders[rightId].exists = true;
9          }
10
11         emit ProposedRestructure(rightId, _getProposedRestructure(
               rightId));
12     }
```

Above is the first step which is the proposal, this is a small function that iterates through all the new shareholders and adds their address and shares to a new mapping called *_newHolders* which is the same as the *_shareholders* mapping and is there to hold proposed ownership structures before they're "bound". An event is emitted telling the back-end the proposal has been saved to the chain and to start accepting votes.

Figure 24: BindRestructure function from CopyrightBase.sol

```
function BindRestructure(uint256 rightId, bool accepted) external
    override validId(rightId) isExpired(rightId) isShareholderOrApproved
    (rightId, msg.sender) payable
{
  _checkHasVoted(rightId, msg.sender);

  // record vote
  _proposalVotes[rightId].push(ProposalVote(msg.sender, accepted));
  _numOfPropVotes[rightId] ++;

  for (uint256 i = 0; i < _proposalVotes[rightId].length; i ++)
  {
    if (!_proposalVotes[rightId][i].accepted)
    {
      _resetProposal(rightId);
      emit FailedProposal(rightId);

      return;
    }
  }

  // if the proposal has enough votes, **** 100% SHAREHOLDER CONSENSUS
      ****
  if (_numOfPropVotes[rightId] == _numberOfShareholder(rightId)) {

    // proposal has been accepted and is now binding

    OwnershipStake[] memory oldOwnership = OwnershipOf(rightId);

    // reset has to happen before new shareholders are registered to
        remove data concerning old shareholders
    _resetProposal(rightId);

    _shareholders[rightId] = _newHolders[rightId];

    delete(_newHolders[rightId]);

    emit Restructured(rightId, RestructureProposal({oldStructure:
        oldOwnership, newStructure: OwnershipOf(rightId)}));
  }
}
```

Now looking at the more complex *BindRestructure* function. First the address is checked if they've voted already, the vote is then recorded in a new mapping *_proposalVotes*, all votes are checked, if any of the votes are false then the whole proposal is rejected, checks if all the shareholders have voted, if all the votes are in set *_shareholders* to the proposed structure from *_newHolders* and emit an event.

This is a point of possible future development or discussion, for a proposal to be *"bound"* a unanimous vote is needed however this system could take advantage of the distribution of shares with only a majority of shares needed to make a change.

I decided to keep the voting unanimous over concerns of complexity (extension of development time was predicted) and a possible moral issues as to the possibilities of exploitation using this. An issue of exploitation is inherent to both implementations

however a unanimous vote gives equal power of exploitation to every party whereas using shares would give more power to high staked parties.

5.1.4 Modifiers

Modifiers are pieces of code run at either the end or start of a function call usually used to verify function parameters, I'm using them exclusively at the start of function calls to test addresses, ids and expiry. The design of a modifier usually consists of an assertion plus any processing needed on the data.

Figure 25: isShareholderOrApproved modifier from CopyrightBase.sol

```
1  modifier isShareholderOrApproved(uint256 rightId, address addr)
2  {
3    uint256 c = 0;
4    for (uint256 i = 0; i < _shareholders[rightId].stakes.length; i++)
5    {
6      if (_shareholders[rightId].stakes[i].owner == addr) c ++;
7    }
8    require(c == 1 || _approvedAddress[rightId] == addr, NOT_SHAREHOLDER)
       ;
9    _;
10 }
```

This modifier authenticates a message sender is allowed to make changes to a specific ***copyright*** by checking it exists in ether *_shareholders* or *_approvedAddress* mappings.

Figure 26: validAddress modifier from CopyrightBase.sol

```
1  modifier validAddress(address addr)
2  {
3    require(addr != address(0), INVALID_ADDR);
4    _;
5  }
```

Ethereum has a reserved address called the *zero-address* only used for ***smart contract*** creation transactions, it's good practice to check all the addresses the contract handles are not the *zero-address*.

Figure 27: isExpired modifier from CopyrightBase.sol

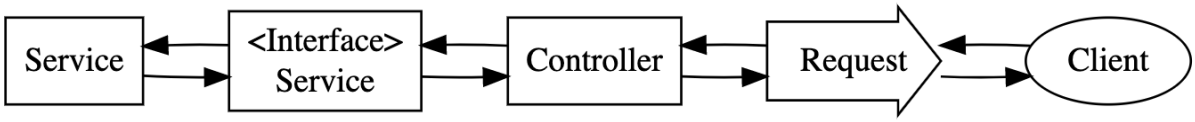
```
1  modifier isExpired(uint256 rightId)
2  {
3    require(_metadata[rightId].expires > block.timestamp, EXPIRED);
4    _;
5  }
```

Copyright expiry is handled with a modifier instead of an explicit state change as it's not feasible to have a timed process that runs when the expiry time is hit because of the time scale ***copyright*** works in. Therefore every time a transaction interacts with a ***copyright*** the expiry time is checked against the current block timestamp throwing an error code if expired.

5.2 Back-end

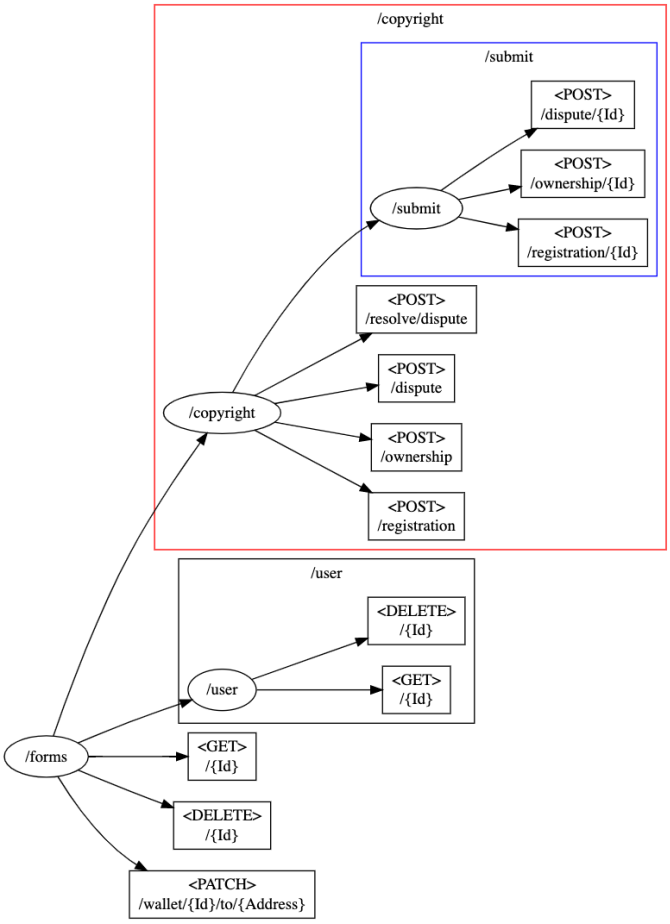
The back-end comprises of an **ASP.NET** API and static file server, the API is comprised of controllers that depend on services which hold business logic.

Figure 28: HTTP request generic data flow



5.2.1 API overview

Figure 29: Forms controller endpoints

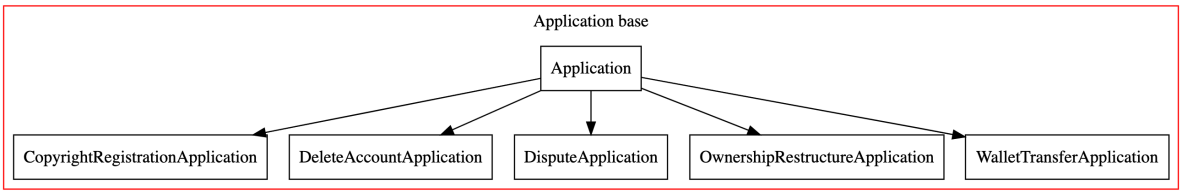


looking at the most complex and largest controller I tried to keep it as *RESTful* as possible with all endpoints describing the resource accessed in conjunction with appropriate HTTP methods describing the action performed.

5.2.2 Applications framework

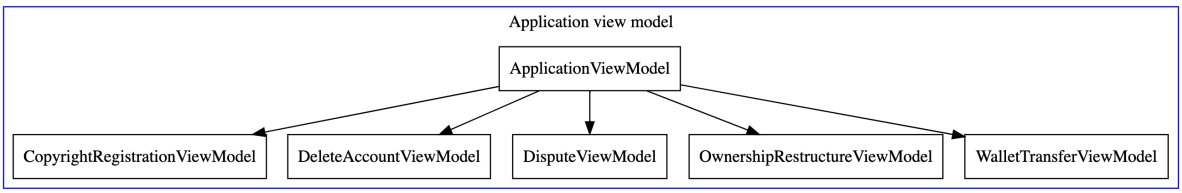
The applications framework exploits object oriented programming techniques mainly inheritance and polymorphism. Application have three classes each an implementation of each abstract base class, this means applications can be handled interchangeably generically and individually based on the child class.

Figure 30: Application base class derived from



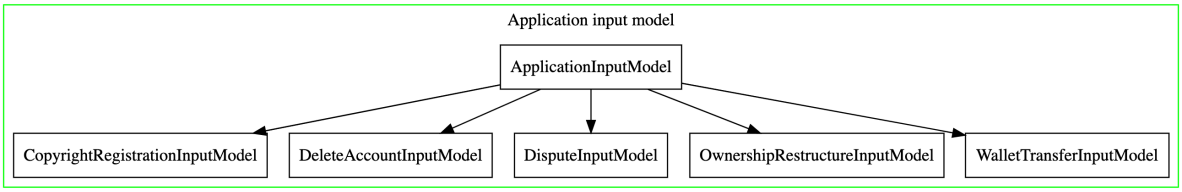
The base *Application* class is used by **EF Core** to generate database migrations creating and modifying tables, therefore this class establishes all relationships in this case a many-to-many with users and one-to-many with a registered work.

Figure 31: Application view model class derived from



Application view models are used when retrieving and displaying data, a lot of the time you don't want to send everything from the base model to the client or some processing/mapping is needed.

Figure 32: Application input model class derived from



Lastly input models that are used when updating applications, this usually represents a real form the user is interacting with. This model is interpreted when updating which modifies the data model on the database.

One huge benefit of all applications using the same base class is that state can be generalised and kept consistent between different types of applications, a completed registration application should logically be the same as a completed dispute application (This state was discussed in design section 4.3.4).

Figure 33: Example application updater CopyrightRegistrationUpdater

```
1  public static class CopyrightRegistrationUpdater
2  {
3      private static readonly List<string> Encodables = new() { "
           OwnershipStakes", "Id"  };
4
5      public static async Task<CopyrightRegistrationApplication> Update(
           this CopyrightRegistrationApplication application,
           CopyrightRegistrationInputModel inputModel, IServiceProvider
           serviceProvider)
6      {
7          var userService = serviceProvider.GetRequiredService<
               IUserService>();
8
9          application.UpdateProperties(inputModel, Encodables);
10
11         if (inputModel.OwnershipStakes != null)
12         {
13             application.OwnershipStakes = inputModel.OwnershipStakes.
                   Encode();
14
15             application.CheckAndAssignStakes(userService, inputModel.
                   OwnershipStakes);
16         }
17
18         return application;
19     }
20 }
```

Each application has an updater which is used for parsing the input model and updating the data model, this one is for the copyright registration application and is fairly simple. First it gets an instance of the user service, calls a static method I made that matches class properties between the input model and data model then updates those properties using the input model, complex object can't be saved in a db column so the ownership stakes are encoded into strings and a relationship is updated/established between all the shareholders and the application.

Figure 34: Example application submitter CopyrightRegistrationSubmitter

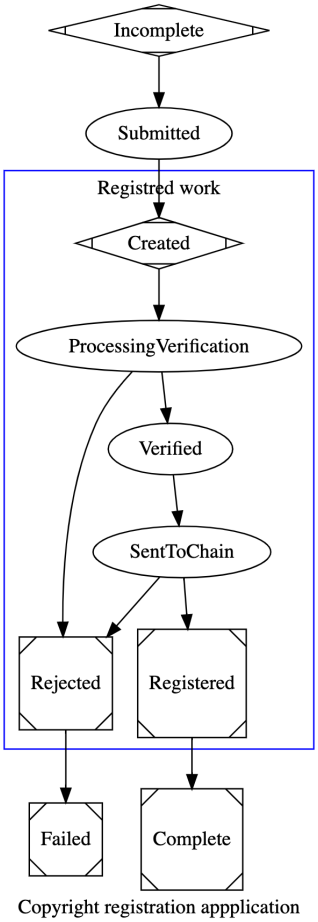
```
public static class CopyrightRegistrationSubmitter
{
    public static async Task<CopyrightRegistrationApplication> Submit(
        this CopyrightRegistrationApplication
        copyrightRegistrationApplication, IServiceProvider
        serviceProvider)
    {
        var registrationService = serviceProvider.GetRequiredService<
            IRegistrationService>();

        await registrationService.StartRegistration(
            copyrightRegistrationApplication);

        copyrightRegistrationApplication.Status = ApplicationStatus.
            Submitted;

        return copyrightRegistrationApplication;
    }
}
```

An application also needs a submitter for when all the data has been inputted, in the case of *CopyrightRegistrationSubmitter* it starts the registration process (creates work and starts verifying) and sets the status of the application to submitted. The majority of submitters start some background process or will require further processing to then be completed.

5.2.3 Registration process

The copyright registration process can be broken down into: fill and submit application, verification, send to chain and transaction verified.

Figure 35: Registration state flow



The first step works as previously discussed, a user fills out a form/application submitting when all data has been inputted. validation of the data is handled on the client side with some server side sanity checks, this was to reduce development workload and complexity however it's compromised by a user calling the API directly either maliciously or if the API was made public.

Figure 36: Finding collisions Line 57:58

```
1 var collision = await Context.RegisteredWorks
2             .FirstOrDefaultAsync(x => x.Hash == work.Hash && x.Id !=
                workId);
```

Verification is handled by a background service working though a queue which checks for collisions by comparing the uploaded work hash against all known work hashes. If a collision is found the application is set to *Failed* and the work is set to *Rejected*.

Figure 37: Register message Line 88:102

```
var register = new RegisterFunction()
{
  To = application.OwnershipStakes.Decode().Select(x => Mapper.Map<
     OwnershipStakeContract>(x)).ToList(),
  Meta = new Meta
  {
    Expires = new BigInteger((DateTime.Now.AddYears(application.
       YearsExpire) - new DateTime(1970, 1, 1, 0, 0, 0, DateTimeKind.
       Utc)).TotalSeconds),
    Title = application.Title,
    Registered = new BigInteger((DateTime.Now - new DateTime(1970, 1,
       1, 0, 0, 0, DateTimeKind.Utc)).TotalSeconds),
    LegalMeta = application.Legal,
    WorkHash = Encoding.UTF8.GetString(application.WorkHash),
    WorkUri = application.WorkUri,
    WorkType = application.WorkType.ToString(),
    Protections = application.Protections
  }
};
```

Once verified a ***copyright*** owner then has to 'complete' the registration which sends a register message (seen above) to the chain returning the transaction hash for later reference.

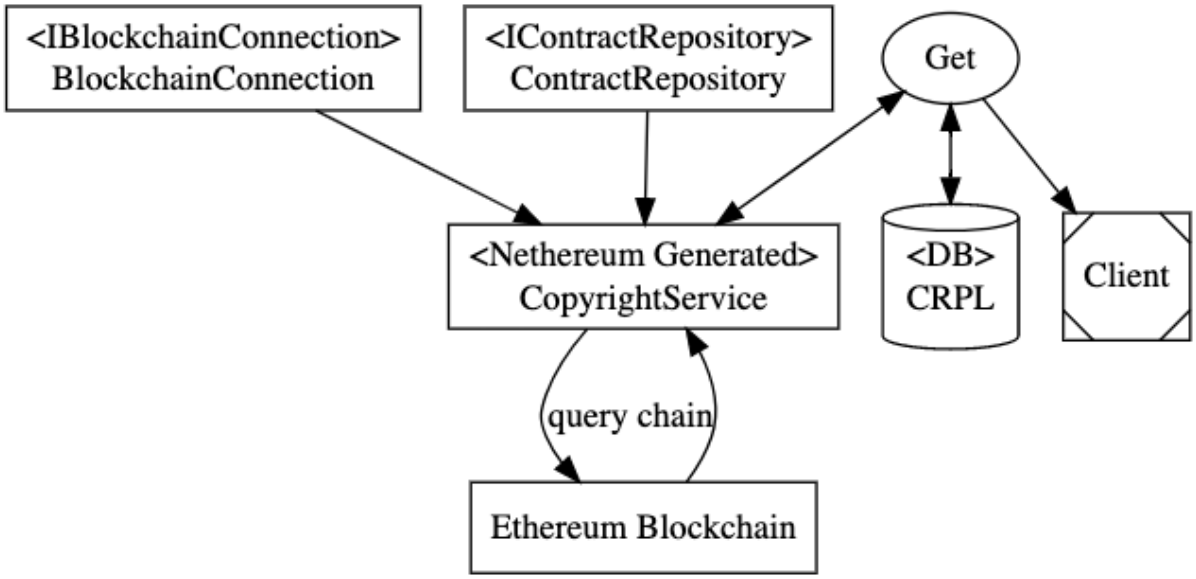
Figure 38: Registered Event Processor Line 29:31

```
context.Update(registeredWork);

registeredWork.RightId = registeredEvent.Event.RightId.ToString();
registeredWork.Registered = DateTime.Now;
registeredWork.Status = RegisteredWorkStatus.Registered;

registeredWork.AssociatedApplication.First(x => x.ApplicationType ==
   ApplicationType.CopyrightRegistration).Status = ApplicationStatus.
   Complete;
```

Nodes on the network will now process this transactions taking an indeterminate time based on how much money you're willing to spend and the current capacity of the network. Once verified onto the chain a *Registered* event will be picked up by the event processors running, the processor will match the transaction hashes setting the work to registered and its registration application to complete.

5.2.4 Queries - Chain injection

Figure 39: Data injection from ***blockchain*** when querying registered works



Because I rely on the ***blockchain*** as the store and point of truth in the system I require a way of querying a ***copyright*** for data to display.

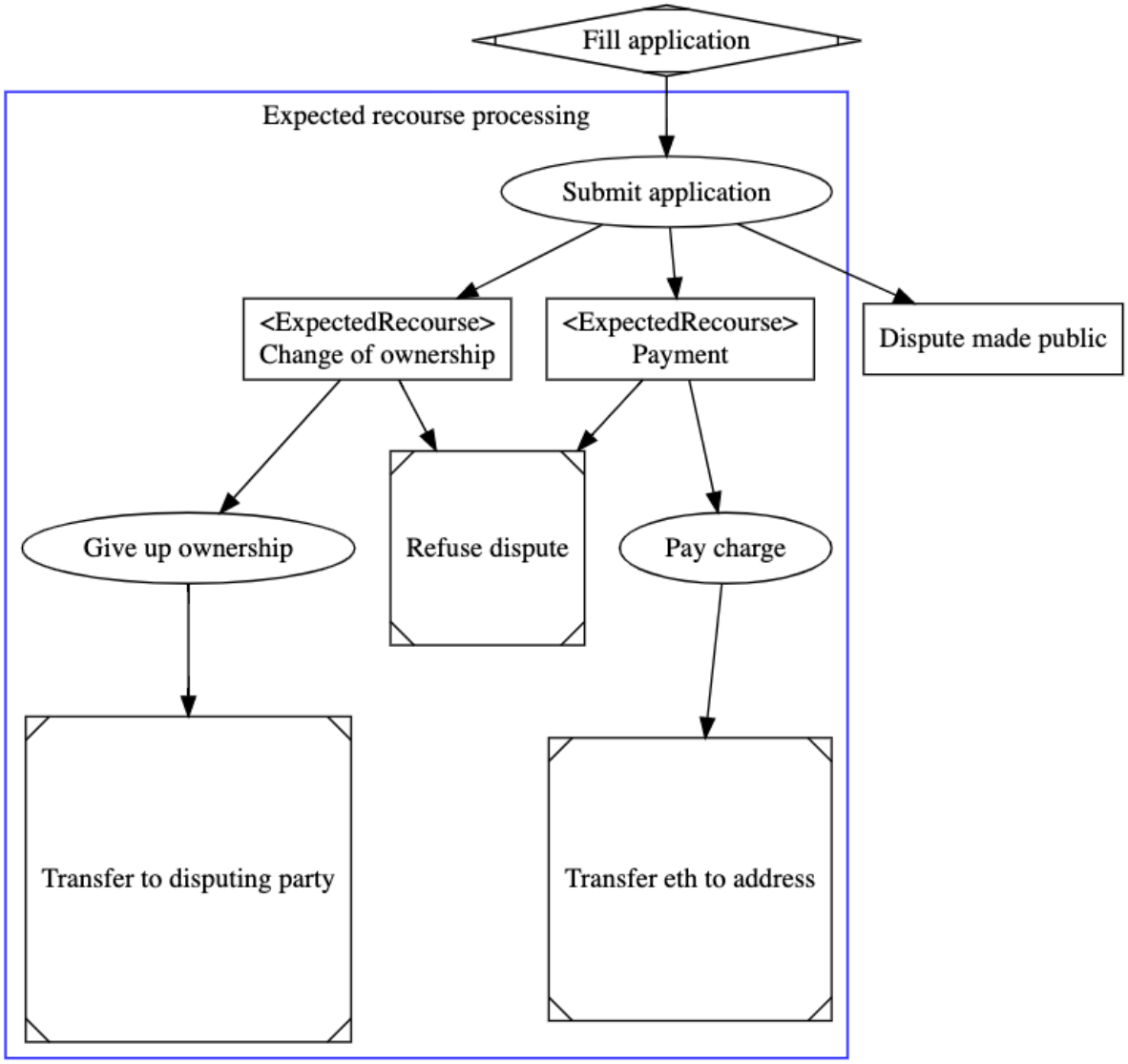
Figure 40: Metadata query from query service Line 155:156

```
1 var meta = await new Contracts.Copyright.CopyrightService(
     BlockchainConnection.Web3(), ContractRepository.DeployedContract(
     CopyrightContract.Copyright).Address)
2      .CopyrightMetaQueryAsync(rightId);
3 ...
4 registeredWork.Meta = meta != null ? meta.ReturnValue1 : null;
```

This is a snippet of a function called *injectFromChain* found in the query service and is called when work is requested from the service, it queries the ***blockchain*** for data then "injects" the result into the existing registered work data model.

5.2.5 Dispute handling

Figure 41: Dispute file and resolve flow diagram



Copyright dispute workflow

Disputes are applications with an *ExpectedRecourse* and will not complete until the attached recourse is handled. Disputes are publicly viewable once submitted to maintain transparency and reduce exploitation.

5.2.6 Blockchain event listeners

To listen for events in the ***blockchain*** **Nethereum** has a class called *Blockchain-Processor* which listens for an event and registers a callback that is runs when that specific event is found.

Figure 42: Registering an event listener for *Registered* event Line 32:33

```
BlockchainConnection.Web3().Processing.Logs
  .CreateProcessorForContract<RegisteredEventDTO>(ContractRepository.
     DeployedContract(CopyrightContract.Copyright).Address, log =>
     EventQueue.QueueEvent(log))
```

I'm using event processors that listen to a specific event type from a specific ***smart contract***, see above this processor is listening for events of type *RegisteredEventDTO* from the deployed contract retrieved from the *ContractRepository*.

All my processors push all events to the event queue to be processed by a service instead of processing in the callback, this increases scaleability and breaks the code down into smaller modular chunks.

Events are pulled from the queue by the EventProcessingService and processed by an EventProcessor which are static classes with one method *ProcessEvent*, one is built for every event type listened to.

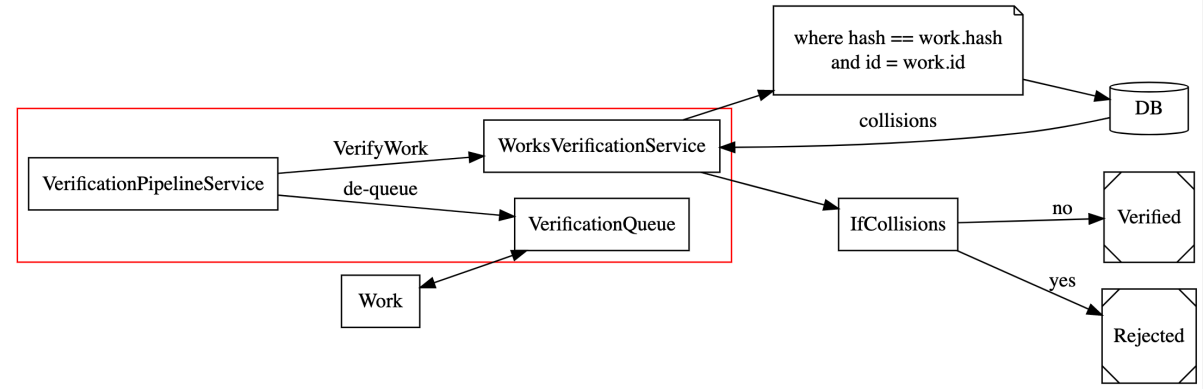
Figure 43: Processing events based on type Line 32:49

```
switch (nextEvent.GetType().FullName)
{
  case var name when name.Contains("RegisteredEvent"):
    await ((EventLog<RegisteredEventDTO>)nextEvent).ProcessEvent(
       ServiceProvider, Logger);
    break;
  case var name when name.Contains("ApprovedEvent"):
    await ((EventLog<ApprovedEventDTO>)nextEvent).ProcessEvent(
       ServiceProvider, Logger);
    break;
  case var name when name.Contains("ProposedRestructureEvent"):
    await ((EventLog<ProposedRestructureEventDTO>)nextEvent).
       ProcessEvent(ServiceProvider, Logger);
    break;
  case var name when name.Contains("RestructuredEvent"):
    await ((EventLog<RestructuredEventDTO>)nextEvent).ProcessEvent(
       ServiceProvider, Logger);
    break;
  case var name when name.Contains("FailedProposalEvent"):
    await ((EventLog<FailedProposalEventDTO>)nextEvent).ProcessEvent(
       ServiceProvider, Logger);
    break;
}
```

5.2.7 Verification pipeline

The verification pipeline uses my background service architecture with a *VerificationQueue* and *VerificationPipelineService* that dequeues and processes verification of a work.

Figure 44: Verification pipeline



When a work is dequeued it is verified using the WorksVerificationService method *VerifyWork* that searches the database for existing works with the same hash. If no collisions are found the works status is updated to *Verified*, if collision are found then the work is set to *Rejected*.

Figure 45: hashing a work Line 146:151

```
1 private byte[] HashWork(byte[] work)
2 {
3   using var hashAlgorithm = SHA512.Create();
4
5   return hashAlgorithm.ComputeHash(work);
6 }
```

For hashing the uploaded work I stream the file into a byte array and compute the hash using **SHA-512** which has 2^{512} total possible hash outputs which is 1.34×10^{154} almost double the number of atoms in the universe at around 10^{82} so the possibility of a collision or exploitation is extremely low.

Although **MD5** is mostly commonly used for very quick file duplicity checks these checks are usually just checksums to check file integrity of downloaded files. In terms of cryptographic security **SHA-512** is far better, **MD5** has been 'broken' for years so should never be used for any sensitive or secure data hashing where as the latest **SHA** algorithms are considered secure and used in conjunction with "salt" for passwords in many systems.

5.2.8 Digital signing

Digital signing is handled at the point of registration by WorkSigners built for specific file types. Additionally there's a universal signer applied to all works and simply concatenates a series of unique bytes to the end of the file stream.

Figure 46: UniversalSigner.cs

```
1  public CachedWork Sign(CachedWork work)
2  {
3    byte[] signature = Encoding.UTF8.GetBytes(Signature);
4
5    return new CachedWork
6    {
7      Work = work.Work.Concat(signature).ToArray(),
8      ContentType = work.ContentType
9    };
10 }
```

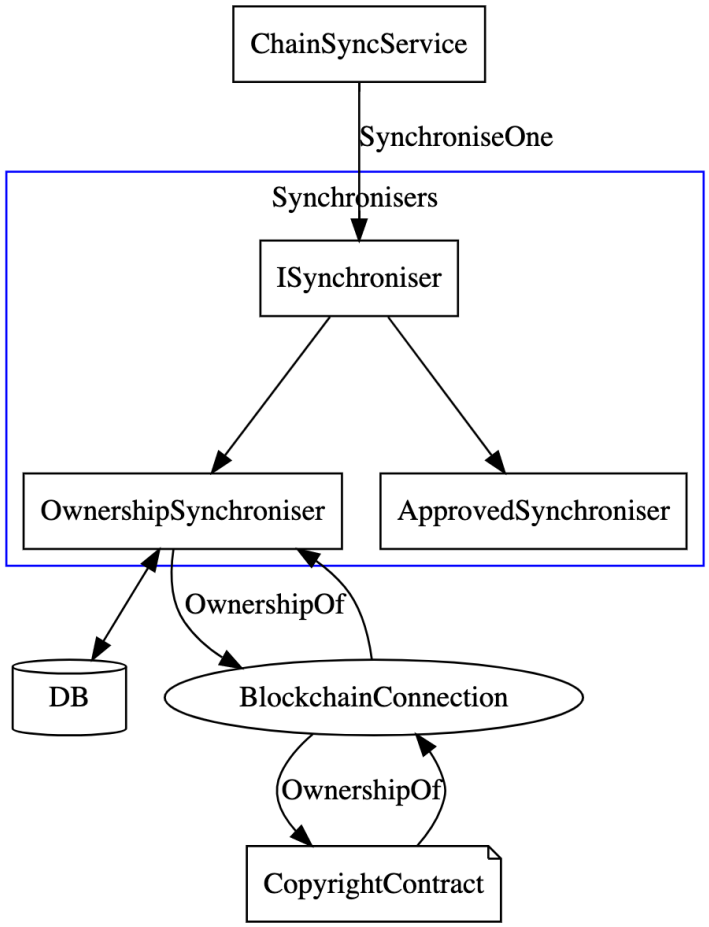
5.3 Database

I've chosen to use a **MySQL** database for this project for two reasons: ease of use and **ACID**. Having previously used **MySQL** for multiple projects in the past plus the brilliant integration between **EFCore**, **LINQ** and **SQL**. **ACID** standing for **a**tomicity, **c**onsistency, **i**solation and **d**urability which ensures data integrity at the cost of speed over a *NoSql* approach like **MongoDB**, it also has support for rigid relationships between entities.

5.3.1 ChainSync

As talked about previously some data has to be kept in parity with the ***Blockchain*** by querying the deployed contract.

Figure 47: ChainSync



Therefore a background service was built that synchronises all the ***copyrights*** currently being tracked with the ***blockchain***, the service checks for changes on startup and then every 24 hours after startup.

Figure 48: 24 hour sync timer Line 25:30

```
1 CronTimer = new Timer(
2   Sync,
3   null,
4   TimeSpan.Zero,
5   TimeSpan.FromHours(24)
6 );
```

There's one synchroniser implemented and it's for ownership structures, it queries the ***blockchain*** using *OwnershipOf* which is a method on the contract that returns an ownership structure for a specific ***copyright***. It then checks against what is saved in the database, if changes are found relationships need to be removed and updated.

Figure 49: ownership similarity Line 71, 77:86

```
1  if (ownerships.Count != work.UserWorks.Count || !work.UserWorks.All(x
       => ownerships.Contains(x.UserAccount.Wallet.PublicAddress.ToLower())
       ))
2  ...
3  work.UserWorks.Clear();
4  ownerships.ForEach(async owner =>
5  {
6    var user = await Context.UserAccounts.FirstOrDefaultAsync(x => x.
         Wallet.PublicAddress.ToLower() == owner.ToLower());
7    if (user == null) throw new UserNotFoundException(owner);
8    work.UserWorks.Add(new UserWork()
9    {
10     UserAccount = user
11   });
12 });
```

5.4 Front-end

A fancy UI was not a high priority as the core back-end and ***blockchain*** interaction is expansive and complex enough in isolation, however I wanted to at least give a user the ability to interact with the system I've built hopefully in a usable and accessible way.

Most user interface design thought and work was spent on the four key forms (user register, copyright register, ownership restructure and dispute) as from previous experience forms are the hardest piece (for me) of interface design by far. To make this easier I counter intuitively at first did not care about the design or usability of the components I was building only getting the data into the form and sent off to the back-end.

This produced some initial designs that functionally work but were confusing users and just didn't look very nice, I've found it's very easy to get hung up on user interface when building systems[iv] which slows down development especially in the early stages of development when the constraints and needs of the system are changing.

Then after all core features had been built I went back to my forms and redesigned all input components using a consistent set of rules, requirements and style.

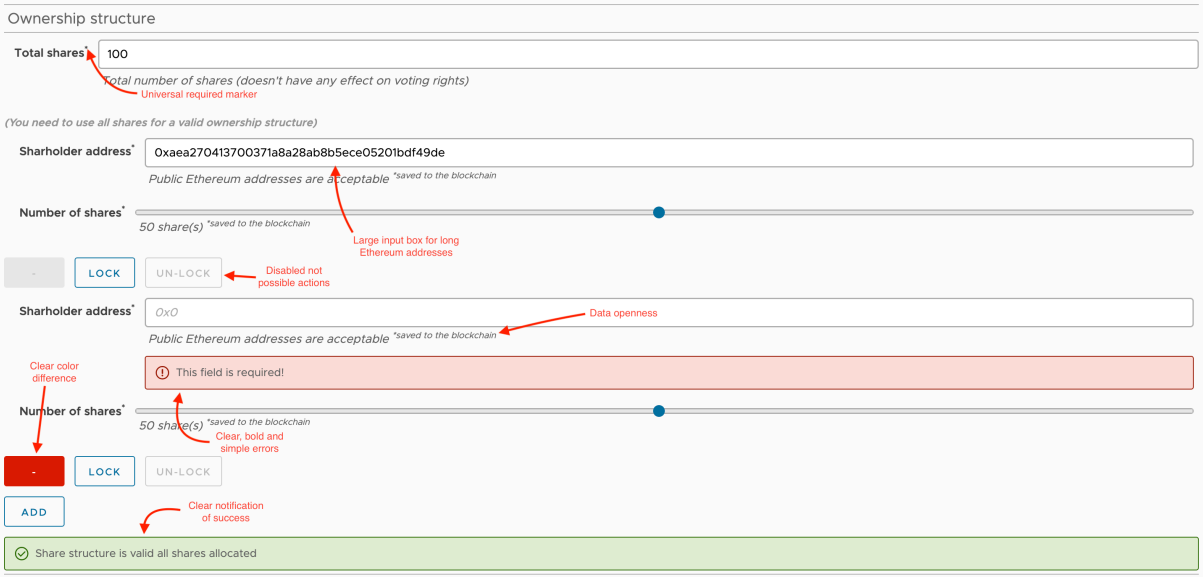
[iv] I think this is down to the fact that it's the single point of contact with your users, most people will not see and or care about business logic code but how your interface looks and feels are front and centre. this would be okay in a real product development setting but for this project it's just not within the scope.

Figure 50: Example input markup cpy-registration-form.component.html 9:18

```
<div class="input-container">
  <label class="input-label">Title<sup>*</sup></label>
  <div class="input-control">
  <input type="text" name="title" formControlName="Title" placeholder="
     Hello world"/>
  <div>This is a searchable title for the copyright, it doesn't have to
      be unique only descriptive and relevant <sup>*saved to the
     blockchain</sup></div>
    <clr-alert class="input-error" [clrAlertClosable]="false"
       clrAlertType="danger" *ngIf="InvalidAndUntouched('Title')">
      <clr-alert-item>This field is required!</clr-alert-item>
   </clr-alert>
  </div>
</div>
```

Below is the my ownership-structure-form component that's used in both cpy-registration-form and cpy-restructure-form, this means all forms have to be consistent across the application so any section of a form will 'fit' into any other form.

Figure 51: Ownership structure form component



6 Discussion

6.1 Limitation

6.1.1 Scope

At every point in this projects design and development limitations of scope have been a constant cause of discussion and often dictated architectural or functional decisions. This is simply because the potential scope of the issue I'm trying to solve is extremely expansive, especially at the intersection of the real legal world and digital constant world. The main example is disputes which are essentially a disagreement over what is 'real', because even though a ***blockchain*** or even just computers alone do keep a perfect record that record can always be *wrong* but so can the real world. Is the computer mirroring the world or enforcing how the world should work?

6.1.2 Implementation

As a result of limited scope the final product is therefore limited in some of its implementation, this is functionality thats been built but hampered by scope and complexity. First is **gas**[v] which for simplicity is covered completely by the system meaning all transaction fees are paid by one account and users registering don't have to worry about paying to register, this is an attractive moral/ethical position but not business decision. Because this point is under discussion it can be seen as a limitation if the intent is for the system to charge for registration or a feature of a completely free and open system providing an essential utility coving all costs.

The live version of this project is deployed to the Ropsten Testnet at 0x427c6335... costing a total of *0.004893532690548546 Ether* and using *2,409,858 gas* which works out to £11.28 (based on a £2,304.45 per Ether). Although this is not representative of costs on **Mainnet**, to calculate this cost I can however use the gas used[vi] in conjunction with the current gas price.

Figure 52: Contract deployment cost at different gas prices based on gastracker

Gas used	Price per Ether
2,409,858	£2,304.45

	Gas price	Fee (Eth)	Fee (£)
Low	12	0.028918296	£66.64
Average	50	0.1204929	£277.67
High	200	0.4819716	£1,110.68

Next is data consistency which is a problem faced by all distributed systems however severely more in a system that's distributed across domains the system doesn't control. This is exactly the situation this system is in, I can't control or own the ***blockchain*** just interact and ask it to do things. This is doubly compounded

[v] gas is the cost in computation to process and complete a transaction on the ***Ethereum*** network which is taken from the senders account at the time of transaction, this is effectively a transaction fee like **Visas** roughly 1% fee on debit transactions but based on work instead of transaction value.

[vi] **Ropsten** uses the same proof-of-work system as **Mainnet** so gas calculations will be very similar.

by the fact that everyone else has the exact level of power and control over the ***blockchain*** as I do, meaning anyone can interact with my ***smart contract*** directly without me. I discussed designing and implementing my *ChainSync* solution for this in 4.4.1 and 5.3.1.

6.1.3 Social consensus/acceptance

Currently ***copyright*** is backed by governments along with all other property law and more importantly has societies faith, trust is the largest barrier to success of this type of system. The question really is why would an author register their work with us over an established institution? This is a question of social change of which I'm no expert, therefore I can only express the benefits of this system (discussed mainly in section 2) as the reason for people to convert. Openness and independence I see as a strong benefit and I think a growing consensus of society agrees with me.

6.2 Blockchain technology

6.2.1 What are NFTs?

NFTs are ***smart contracts*** that provide an interface for some immutable data (most commonly digital assets) on a ***blockchain***, that interface includes ownership tracking, ownership transferring and access control management. The keyword here is *non-fungible* which simply means not replaceable by another, an easy analogy for this is a *bag of wheat* and a *final project report*. The bag of wheat is fungible because one is easily replaced by another whereas you cant replace one final project report with another because they're unique to the project.

An NFT ***smart contract*** simply allows a user to save this non-fungible (unique) asset and assign themselves as the owner. This does not inherently mean the user who owns an NFT "owns" the asset unless specifically stated.

6.2.2 Are these NFTs?

Yes and no.

Yes I took heavy inspiration from the EIP-721 or NFT standard when developing my ***smart contract*** and yes the data they represent is definitely non-fungible (copyrightable works are by definition non-fungible).

No my ***smart contract*** is not compatible with the EIP-721 standard and therefore existing NFT products, NFTs are single owner while CRPL supports multiple owners. More importantly I believe the spirit of my contract is in contrast with NFTs which are being used overwhelmingly in *get rich quick* schemes, fraud all with a heavy focus on money/value. The goal of my contract is to give protection of work to creators, at no point is money a direct focus other than protecting the value of a persons work.

6.2.3 How long will the blockchain last?

The realist answer to this would be; probably not forever and almost definitely less time than a single ***copyright*** protection (100 years in this case). This is a pretty bleak outlook for my system, ***blockchain*** is still an infant technology even though by tech standards should be days past by now. Just like AI and machine learning

I believe ***blockchain*** is at an incredible height but also a make or break point, either the technology is completely accepted and integrated into the world or it's abandoned never to be seen again. After working with the technology for a couple months now I would bet on the latter outcome over engineering concerns but I've been wrong in the past.

7 Evaluation

7.1 Process

Evaluating the process of designing and building a system.

7.1.1 Design/Pre-Development

Software design and the ratio of time spent on design over development is a debated topic within the industry, the most popular theory currently being *Agile* which is biased towards development over design whereas *Waterfall* considered a legacy methodology believes in designing the entire solution before development can take place.

I focused my design on gaining knowledge of the technologies I had no previous experience: ***blockchain*** interaction and ***smart contract*** design. Only abstract models of the system were drawn so I could research all the needed technology.

In practice this strategy was reasonably successful as the overall architecture is representative of my original design, in spite of this many key systems were underestimated or poorly designed largely thanks to lack of knowledge. Although this is how agile development is supposed to work, it's okay if fundamental change happens in fact you should always consider change and change quickly.

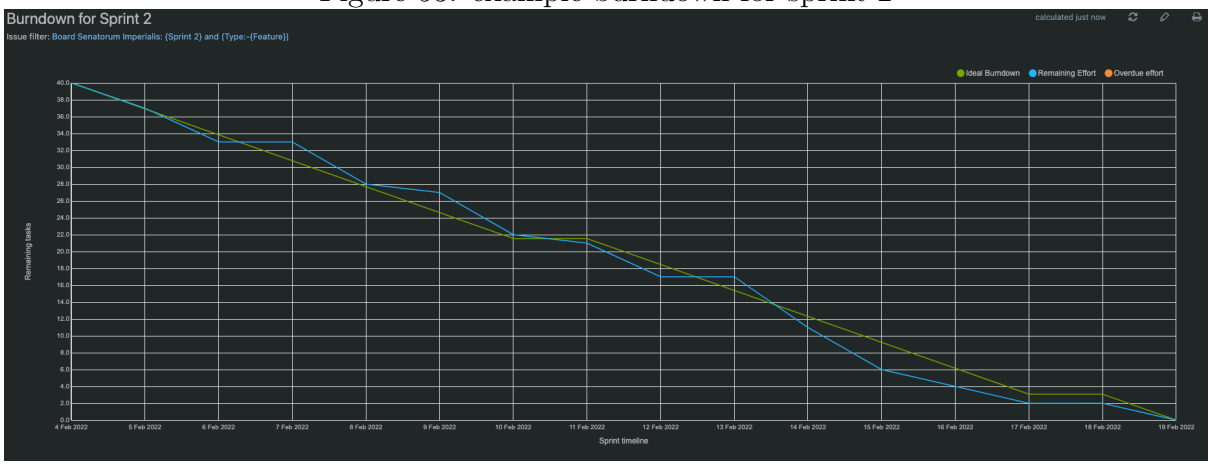
7.1.2 Development

As far as timing development was perfect, every deadline was hit on time with no need for extension or reevaluation. I put this down to aggressive and continuous scope handling, it's very easy as an engineer to completely over-engineer a system and I've had problems managing this desire in the past. However for this project I clearly defined the scope in my mind so whenever new complexity was being debated I always forced myself to consider scope and time constraints.

As a result if you read all my sprint reviews and burndown graphs I consistently hit deadlines and manage the amount of work assigned to each sprint.

In conjunction with agile development I also initially planned to use a development workflow called **TDD** (**T**est **D**riven **D**evelopment) that says functionality should first be test cases the developer then aims to pass by implementing functionality. It's very popular in the *Agile* space and can increase code quality and speed up development, however functionally limited when used in larger user focused and very infrastructure heavy systems. I still tested heavily but my implementation wasn't dependent on testing.

Figure 53: example burndown for sprint 2



7.1.3 Was it agile?

To assess if my process has been agile I'll being using the 4 values from the agile manifesto[5]

Individuals and interactions over processes and tools isn't very applicable as this is an individual project.

Working software over comprehensive documentation links back to my troubles with **TDD** I was focusing more on passing the test than what the software need to do.

Customer collaboration over contract negotiation is an area I could've done a lot more in, my customer is an imagined aggregate of certain struggles not a real person this could've been a better representation with more hands on research.

Responding to change over following a plan would be a great summary of this project because although I had a fairly comprehensive plan I only every took it as a guide, If I'd spent weeks on a master plan it would've only been a waste of time.

7.2 Product

Evaluating the system that has been built functionally and non-functionally.

7.2.1 Functional specification

Have I built a system that satisfies my specification?

Figure 54: success of each functional requirement

Feature	Successfully implemented?
Copyright smart contract	A fully featured copyright contract has been built and deployed onto a test blockchain, the contract is capable of registering a copyrighted work with sufficient metadata to establish ownership and specific legal protections.
Multi-party distribution	Ownership of a registered work is represented as a multi-party share structure allowing many authors to have "ownership" of a single work.
Ownership transfer	The ownership of a registered work can be changed via a proposal and vote system requiring the unanimous consensus of all current owners. *Voting is not majority share-based like limited companies.
Work verification	Verification of works is currently simple using hash comparison, this finds complete file accurate copies and so therefore even simply adding a zero or null data to the file bypasses verification. To solve this problem more complex algorithms will be needed or the use of third-party services, this was outside the scope of this project.
Dispute filing	The system allows anyone (except the owner) to dispute a registered work for a given reason and the choice of an expected recourse (ownership change or payment). Right owners can then either reject the dispute or accept and apply the expected recourse.
Digital signing	Once a work has been registered we inject metadata into the uploaded file for proof of registration, this is done two ways: custom singers built for supported file types and a universal signer that inserts data at the end of any file type.
Decentralised Work CDN & proxy	All registered works are automatically saved to IPFS (InterPlanetary File System) which is a peer to peer network for storing files distributed securely in chunks over multiple nodes. A user can then access this file either through an official gateway (ipfs.io/ipfs/), running your own node to connect to the network or my public gateway (ipfs.harrisonbarker.co.uk/ipfs).
Websocket updates	All updates are done through a single WebSocket, users are subscribed to works and applications when they click on them or have ownership associated with them. The implementation of this is crude though as it was an extra feature added minimally towards the end of the development cycle with many concerns of scaling issues.

7.2.2 Is it fit for purpose

Have I built a system that addresses my issues with ***copyright*** discussed in subsection 2.1? My stated three factors were *complexity, jurisdiction dependence and lack of digital computerised systems*, does my system solve any of these?

First is complexity, by removing as much legal jargon from the process and a customisable protections system users know exactly how their work is protected without the need for legal council or manager therefore empowering creators.

Next is jurisdiction dependence which is an escapable fact of the legal and governmental world that can cause issue and confusion when faced with a global inter-

net culture. My system is global both because of ***Ethereum*** and its independence from any government.

Lastly digitalisation/computerisation, my system is computerisation of ***copyright*** law which solves my last two points but more importantly it's built for copyrightable digital work which traditional ***copyright*** is lacking in. Standardised digital signing and verification is proof of this ambition.

7.2.3 Testing

The system has been extensively unit tested at three levels: ***smart contract***, back-end and front-end middleware.

Because contracts run within the ***EVM*** a testing environment is needed in this case Hardhat which allows me to deploy and run tests on a ***smart contract***.

Figure 55: Register.ts unit test snippet

```
1  it('Should REVERT no shareholders', async function ()
2  {
3    await expect(deployedContract.Register([], meta)).to.be.revertedWith(
       'NO_SHAREHOLDERS');
4  });
5
6  it('Should REVERT when invalid shareholders', async function ()
7  {
8    await expect(deployedContract.Register([{owner: ethers.constants.
       AddressZero, share: 1}], meta)).to.be.revertedWith('INVALID_ADDR')
       ;
9  });
```

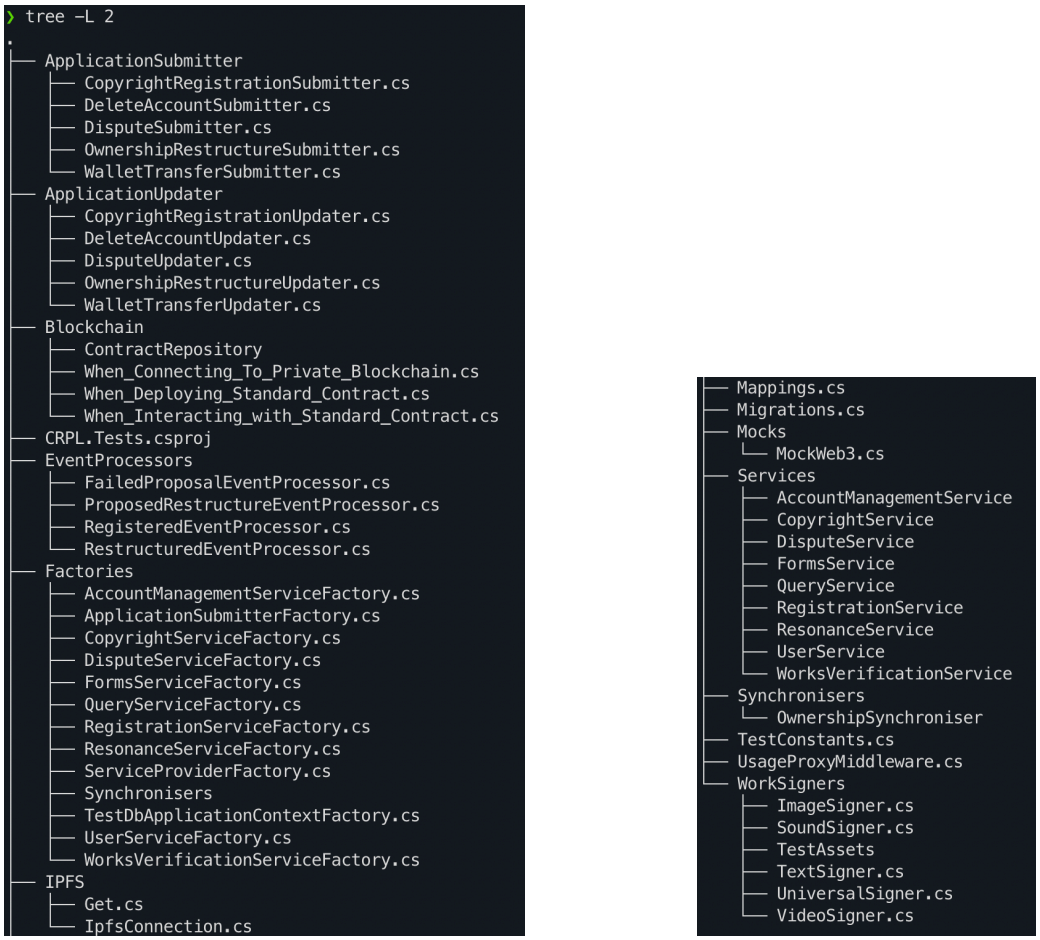
This is what the tests look like there're then run outputting statistics for each payable transaction on the contract including deployment. all ***smart contract*** test can be found here.

Figure 56: contract statistics from Hardhat

Solc version: 0.8.4	Optimizer enabled: true	Runs: 200	Block limit: 30000000 gas
Methods			
Contract	**Method**	**Min**	**Max**	**Avg**	**# calls**	**eur (avg)**
Copyright	ApproveManager	–	–	46284	3	–
Copyright	ApproveOne	36167	38885	37526	4	–
Copyright	BindRestructure	93328	148618	119625	11	–
Copyright	ProposeRestructure	132895	180211	143147	12	–
Copyright	Register	230869	1079035	325002	39	–
Deployments					**% of limit**	
Copyright		–	–	2428208	8.1 %	–

Back-end tests are written in the CRPL.Tests project separated by domain all using NUnit test framework and runner.

Figure 57: Back-end test directory structure



The front-end logic is tested using Angular's built in test framework Jasmine found in *.spec.ts* files.

Figure 58: auth.service.spec.ts nonce fetch test

```
it('should fetch nonce', inject(
      [HttpTestingController, AuthService],
      (httpMock: HttpTestingController, authService: AuthService) =>
      {
        let mockNonce: string = "TEST NONCE";
        service['Address'] = 'TEST ADDRESS';

        authService.fetchNonce().subscribe((nonce: string) =>
        {
          expect(nonce).toEqual("TEST NONCE")
        });

        let request = httpMock.expectOne("user/nonce?walletAddress=TEST
           %20ADDRESS");

        expect(request.request.responseType).toEqual('json');
        expect(request.cancelled).toBeFalsy();

        request.flush(mockNonce);
      }
    )
  );
```

All test reuslts can be found in appendix 10.6

8 Future development

Some potential future development avenues

Royalty payments Royalty payment in cryptocurrency based on the number of shares a user owns in the copyright, automatic based on usage and fairly distributed.

Marketplace An open market for purchasing the legal right to use a work from the rights holder directly.

External/Advanced verification Currently work verification/authentication is very basic and can be exploited, proving authorship is a hard but essential issue to solve.

Analytics provide users with data describing the use of their work.

9 Conclusion

The aim of this project was to successfully represent basic ***copyright*** for works on a blockchain while also providing a web application allowing users to interact and maintain ***copyrights***. This system proves the possibility of such a system through its functionality, however it's not a complete system ready for market as the legal and social complexities of the problem are expansive that need to be taken seriously and a level of scrutiny not possible in the defined scope.

So to conclude while the technical operation of this project fulfils the stated specification it doesn't prove that a blockchain representation of ***copyright*** is the *correct* solution but stand as compelling evidence towards assessing ***blockchain*** as a viable solution.

10 Appendix

10.1 Acknowledgements

Name	Use		Subsystem
Rider	Primary IDE for development	link	DEV
YouTrack	Issue tracking software	link	DEV
VSCode	Editor used for developing smart contracts	link	DEV
RemixIDE	Web based IDE for developing, testing, deploying and interacting with smart contracts	link	DEV
Azure DevOps	CD/CI pipeline for running tests and deploying to Azure	link	DEV
Azure	Cloud based deployment	link	DEV
Ethereum	Public open source blockchain software with smart contract support	link	BLOCK
Solidity	Smart contract programming compiler	link	BLOCK
Metamask	Browser extension based Ethereum wallet with over 30 million users	link	BLOCK
Hardhat	Smart contract development environment	link	BLOCK
Waffle	Smart contract testing framework	link	BLOCK
ASP.NET	Web application/service framework for .NET	link	BACK-END
.NET	Cross platform development framework for C#	link	BACK-END
EFCore	Object database mapper for .NET	link	BACK-END
Nethereum	Ethereum interaction library for .NET	link	BACK-END
NUnit	.NET test runner	link	BACK-END
AutoMapper	.NET object mapping library	link	BACK-END
Cronos	Cron job handling library	link	BACK-END
iTextSharp	PDF interaction and modification library used for digital signing	link	BACK-END
TagLibSharp	Exif tag library used for digital signing	link	BACK-END
ImageSharp	Image metadata library used for digital signing	link	BACK-END
TypeScript	JavaScript superset providing strong typing	link	FRONT-END
Angular	JavaScript framework for development of single page dynamic web applications	link	FRONT-END
Clarity Design	CSS and JavaScript design framework	link	FRONT-END
RxJs	Library for asynchronous JavaScript programming	link	FRONT-END
Signalr	Web-socket platform	link	FRONT-END
web3.js	Ethereum JavaScript API library	link	FRONT-END

10.2 Operational diagrams

Figure 59: Copyright registration sequence

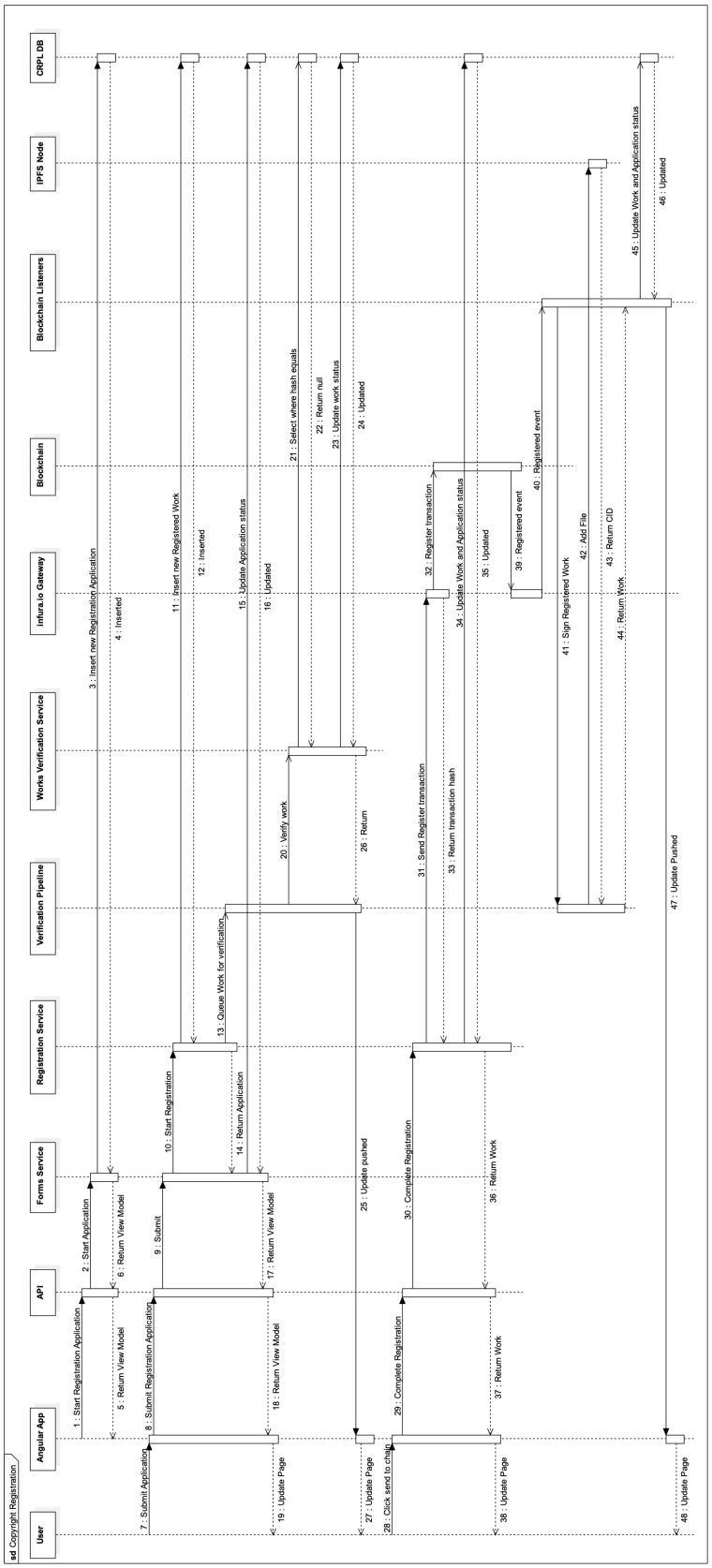


Figure 60: Ownership restructure sequence

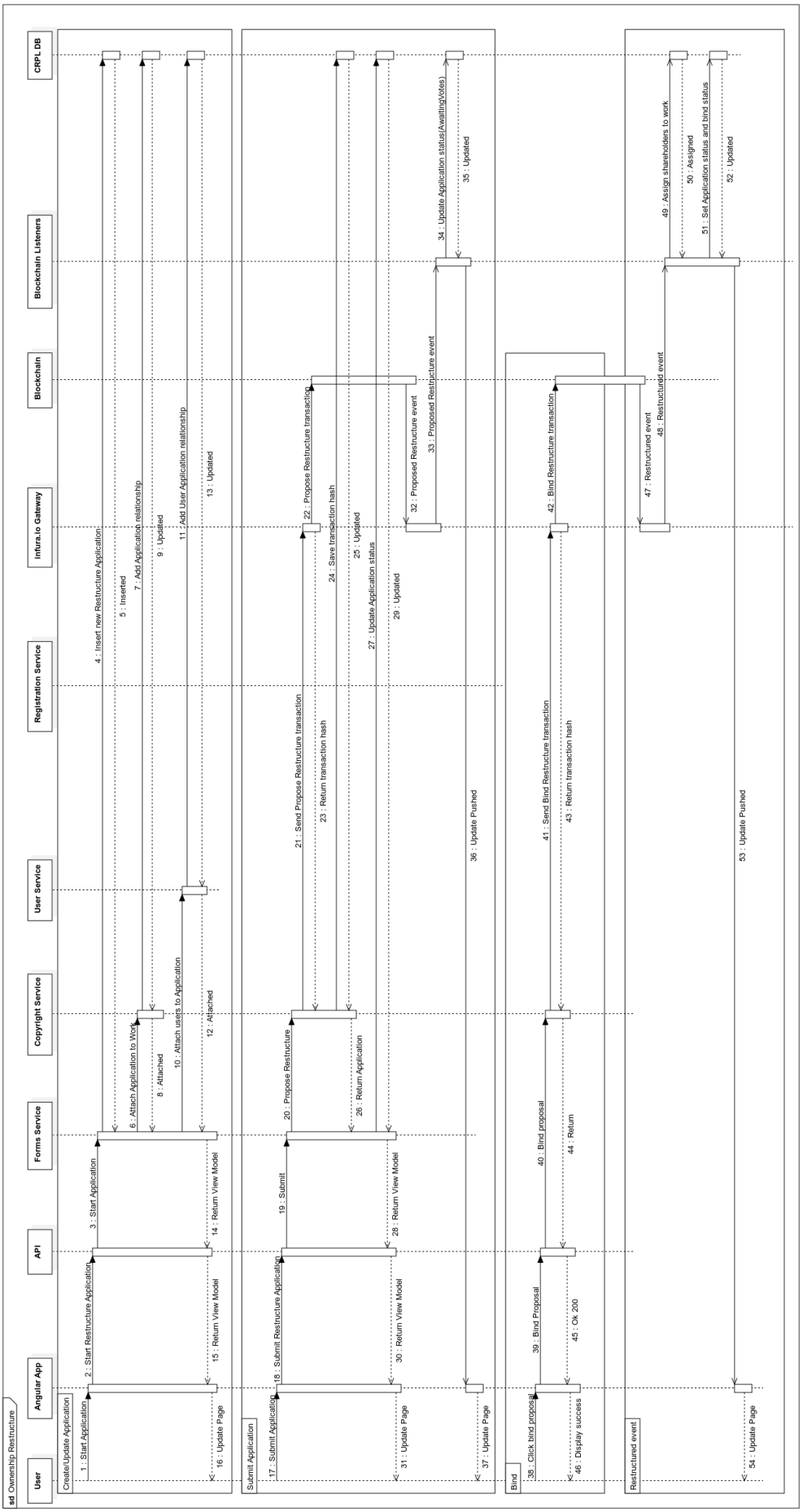


Figure 61: Dispute with payment expected recourse sequence

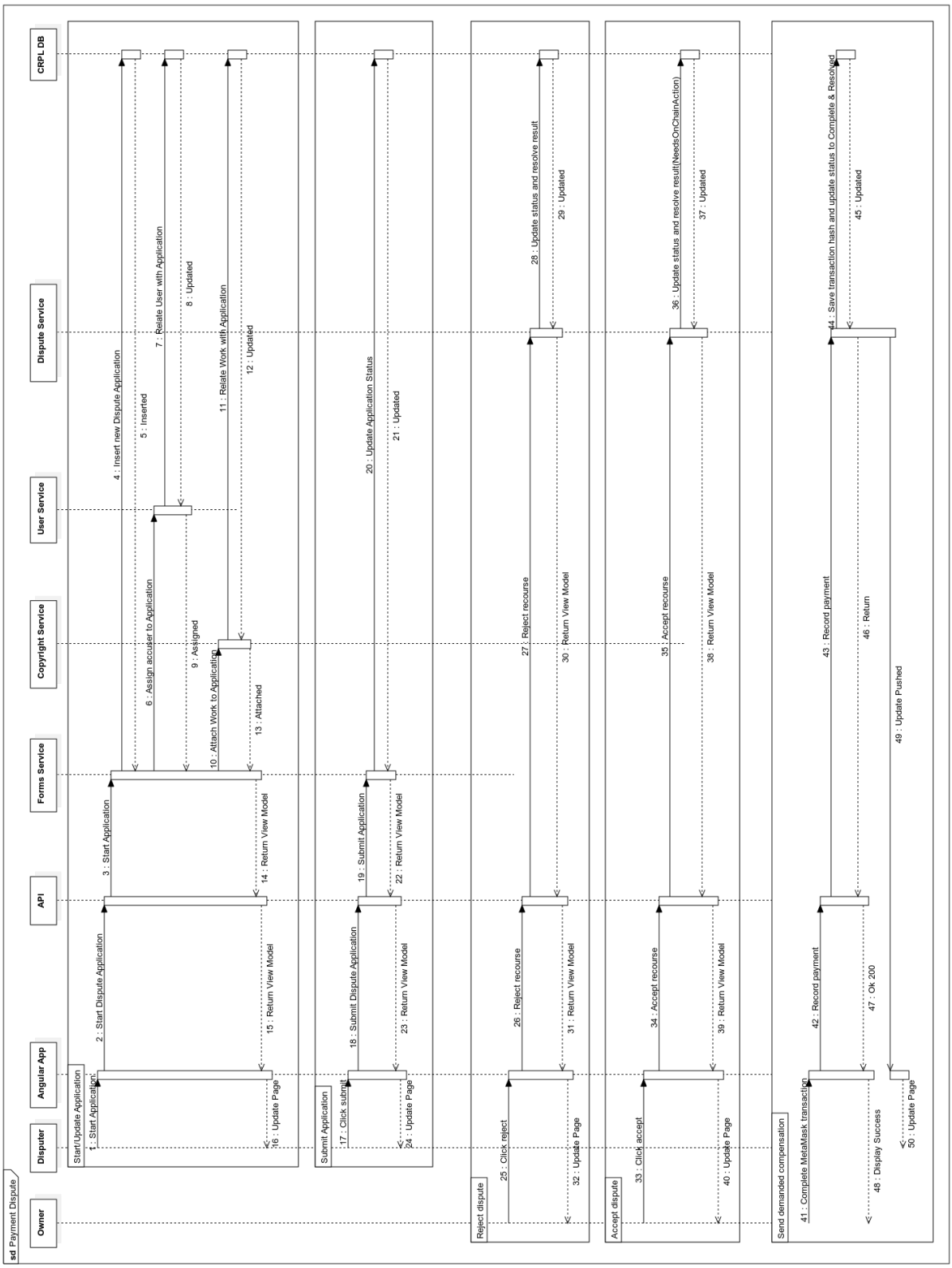


Figure 62: User Authentication (Login/Create Account) sequence

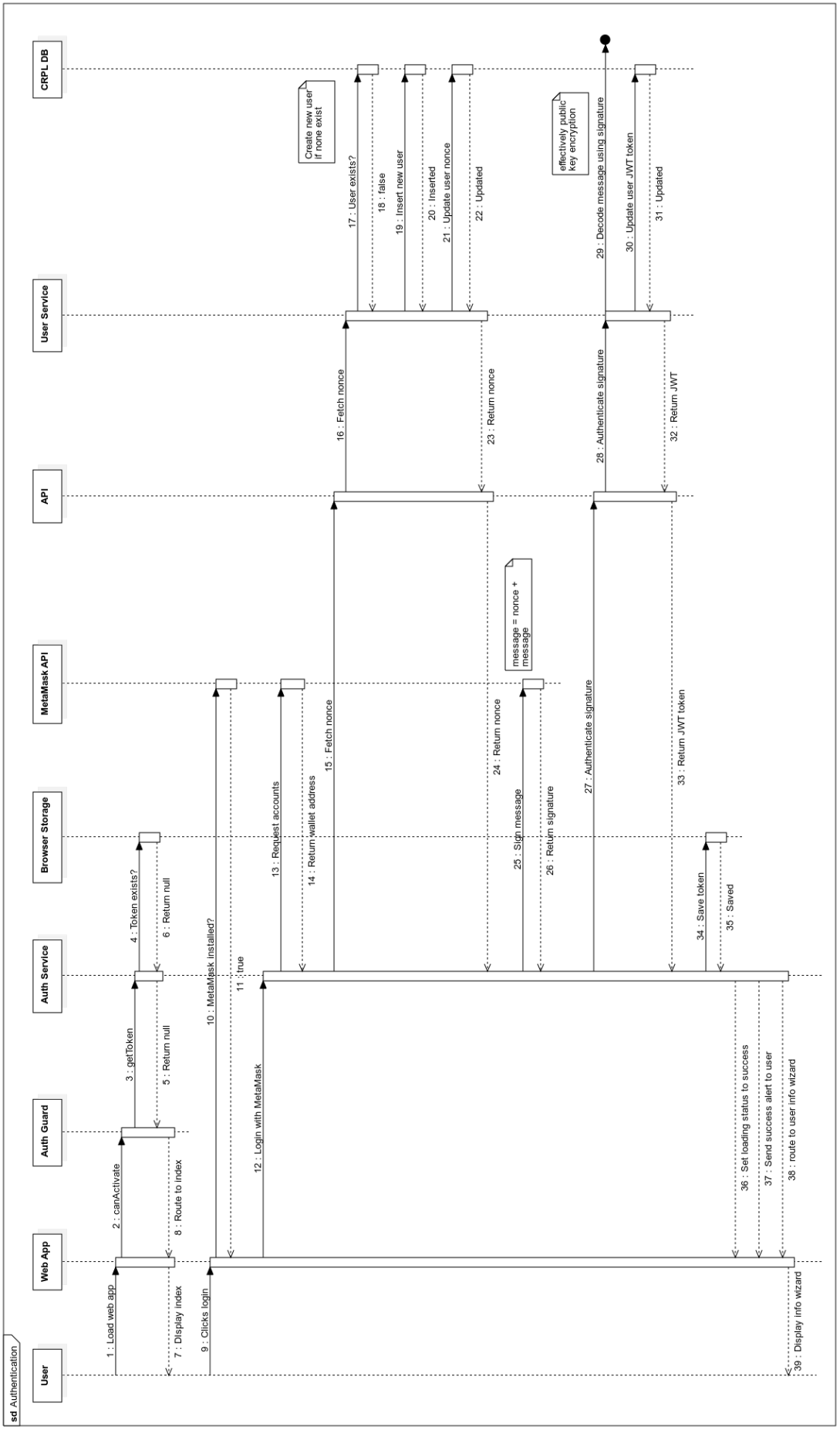


Figure 63: Angular Guarding of pages sequence

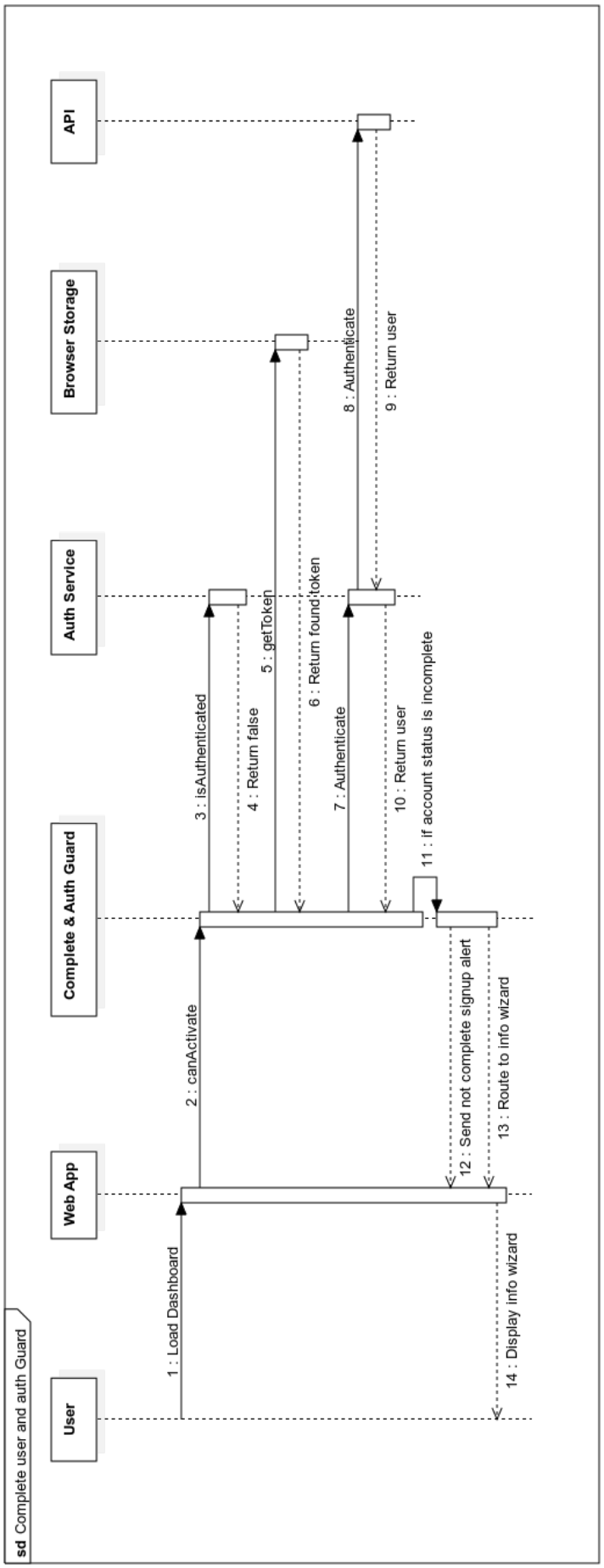


Figure 64: Blockchain event processing sequence

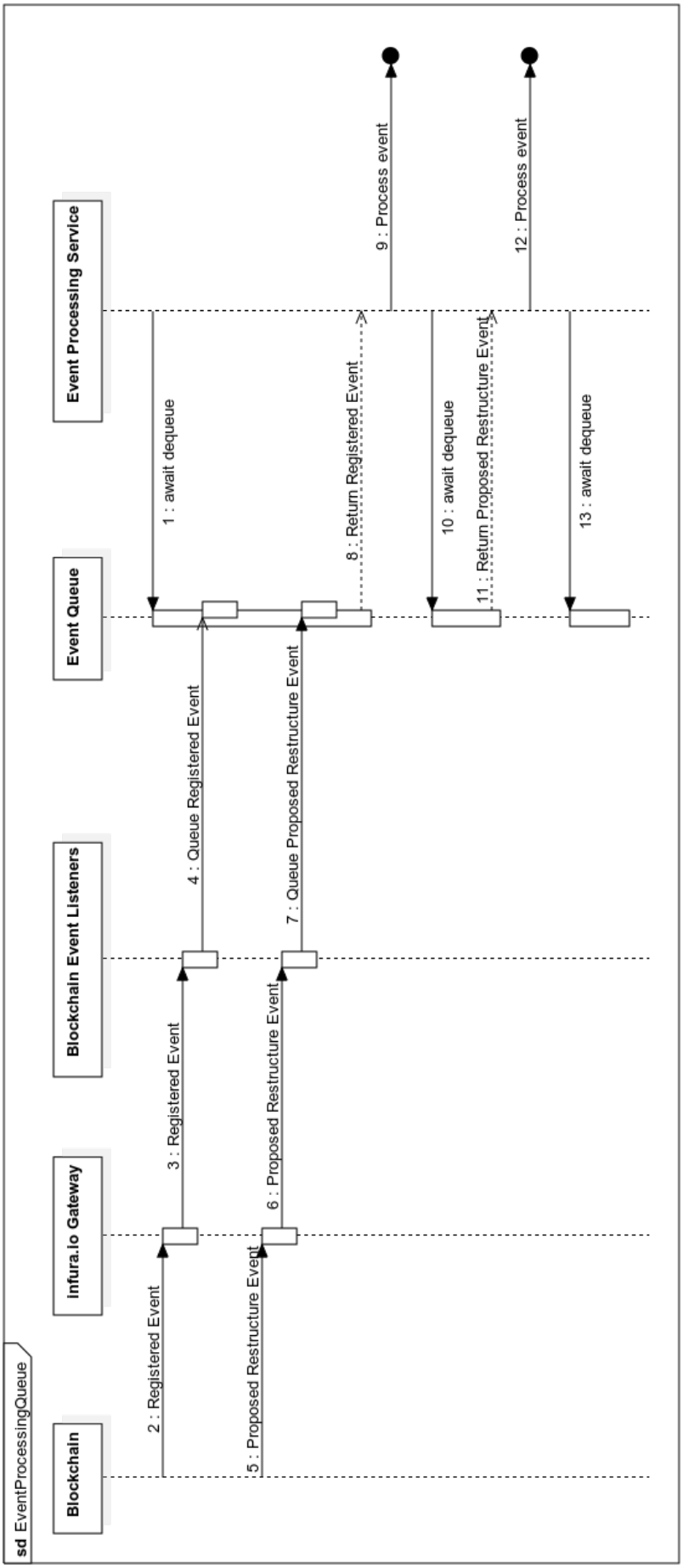


Figure 65: Smart contract Register function flowchart

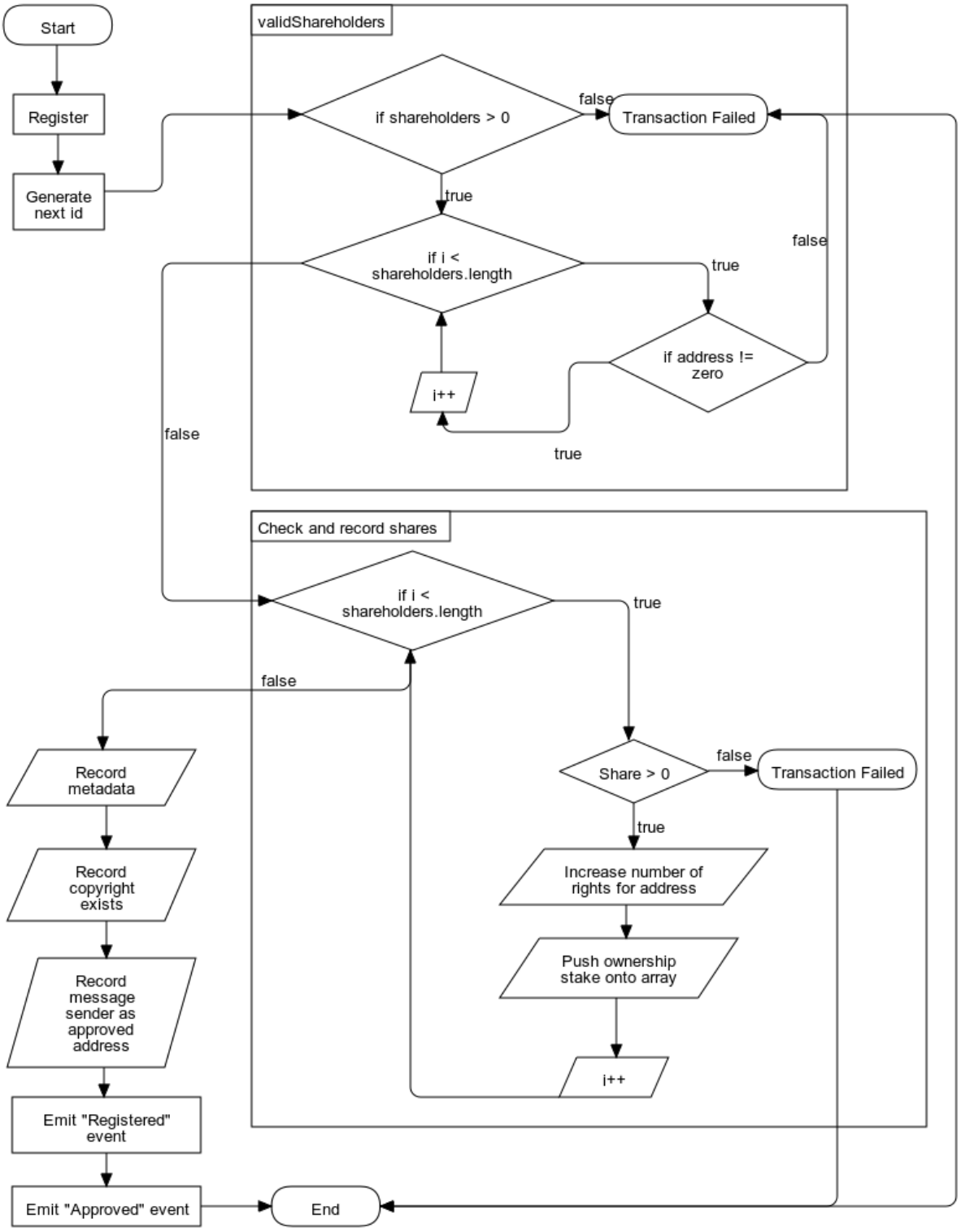
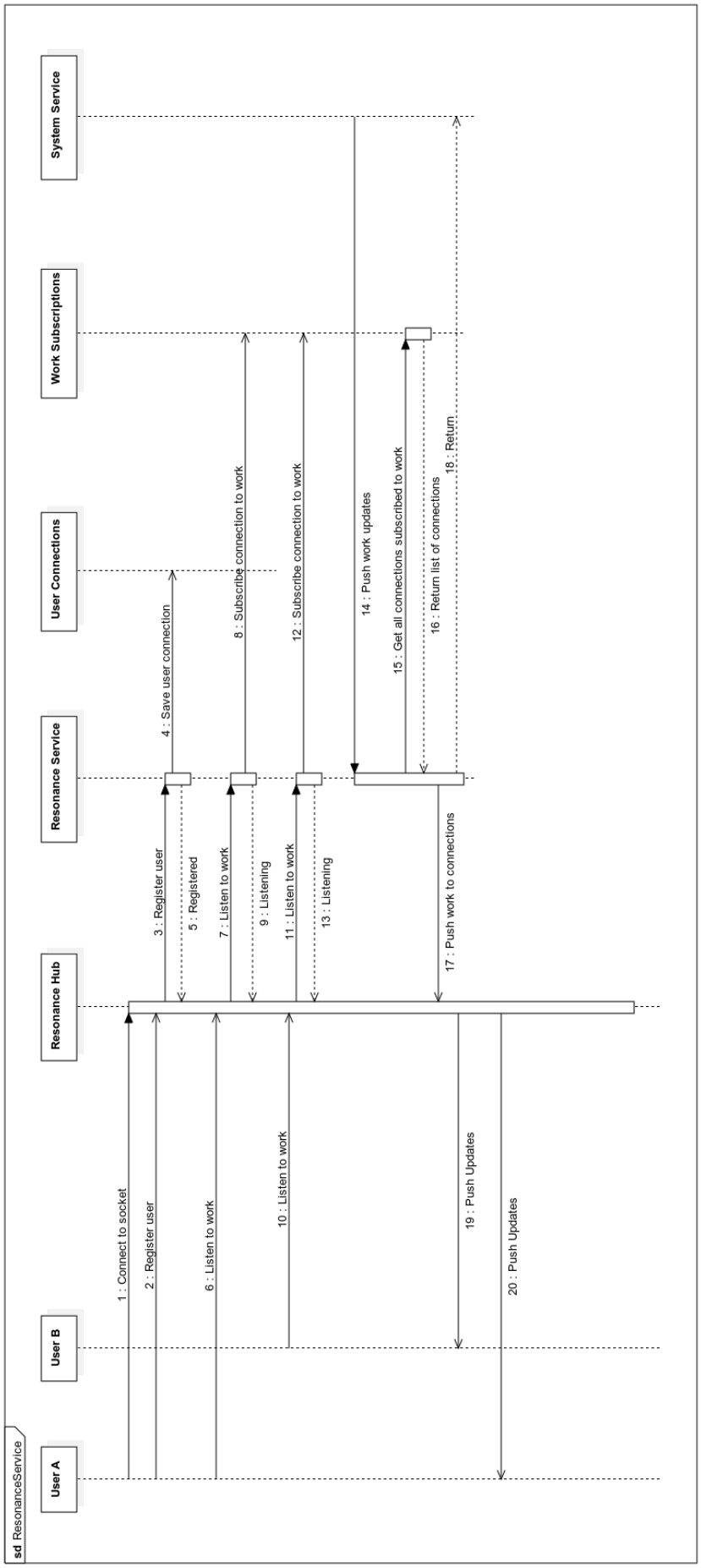


Figure 66: Resonance service (WebSocket service) sequence



10.3 Design docs

Figure 67: Original index page wireframe

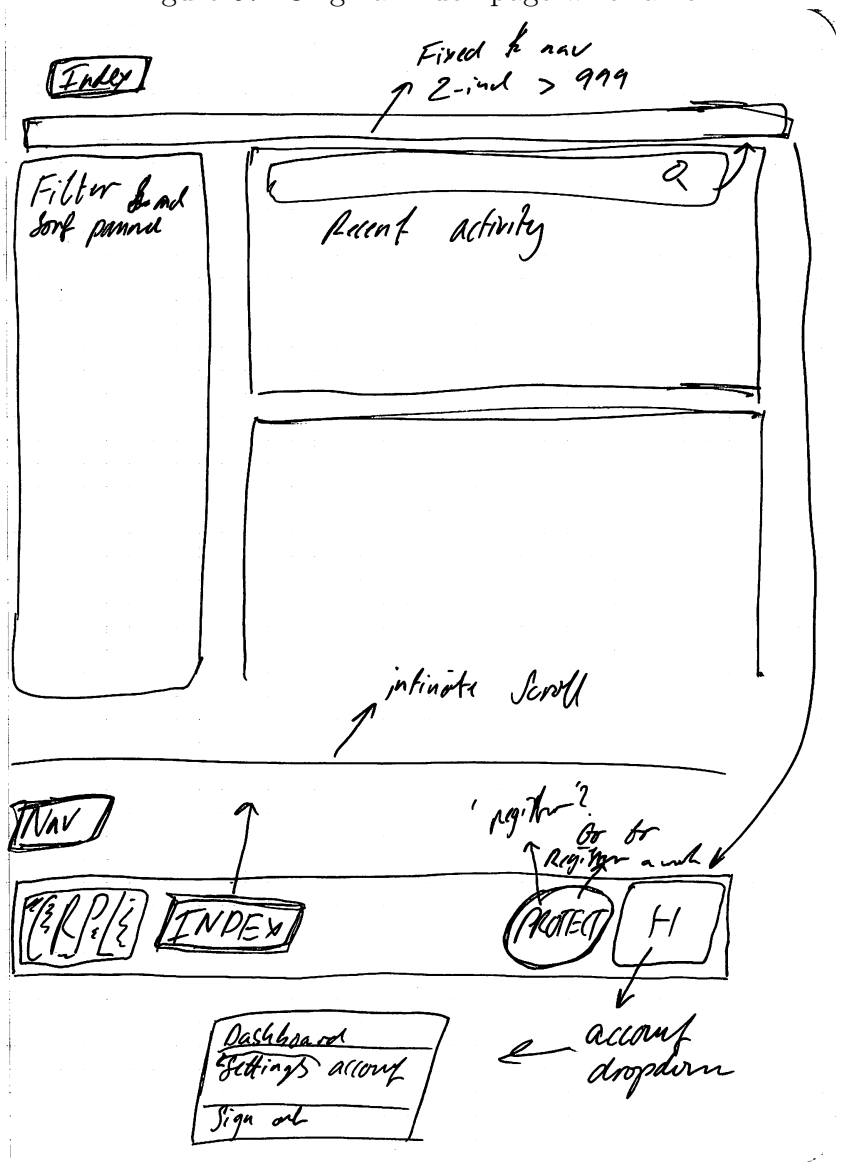


Figure 68: Original dashboard page wireframe

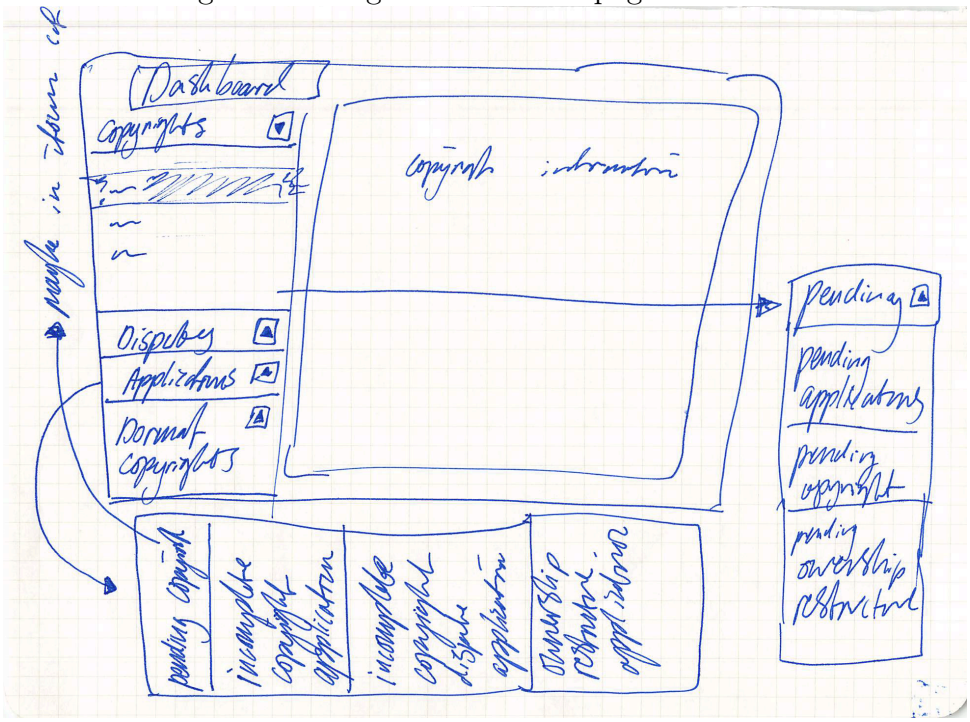


Figure 69: Early system architecture diagram

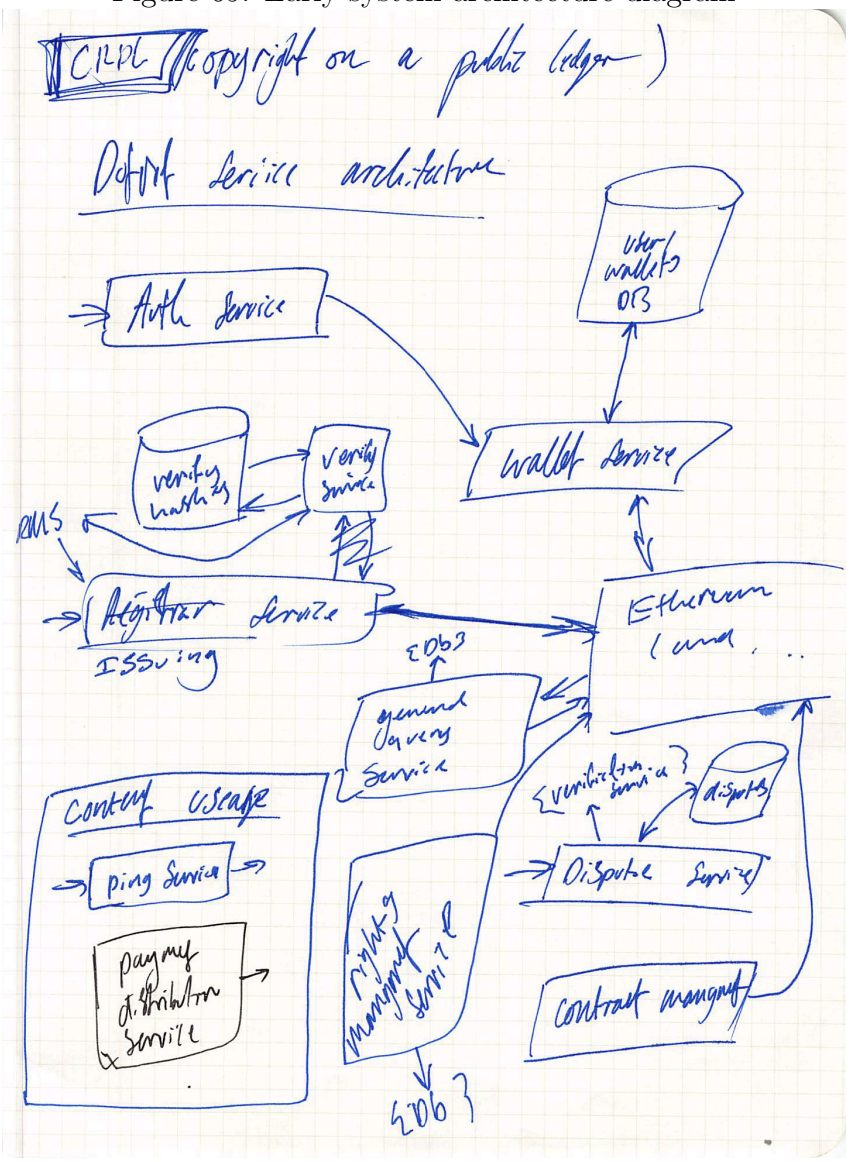


Figure 70: Copyright registration state flow diagram

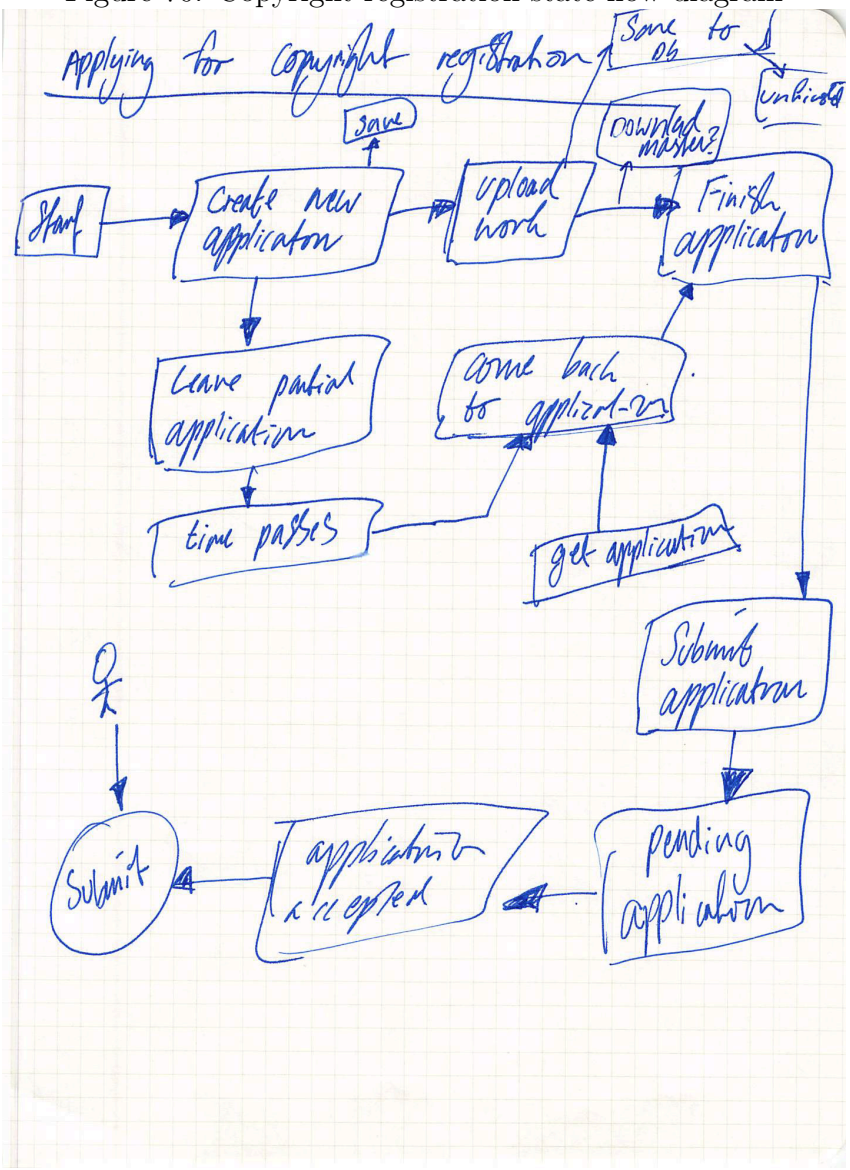


Figure 71: Ownership restructure state flow diagram

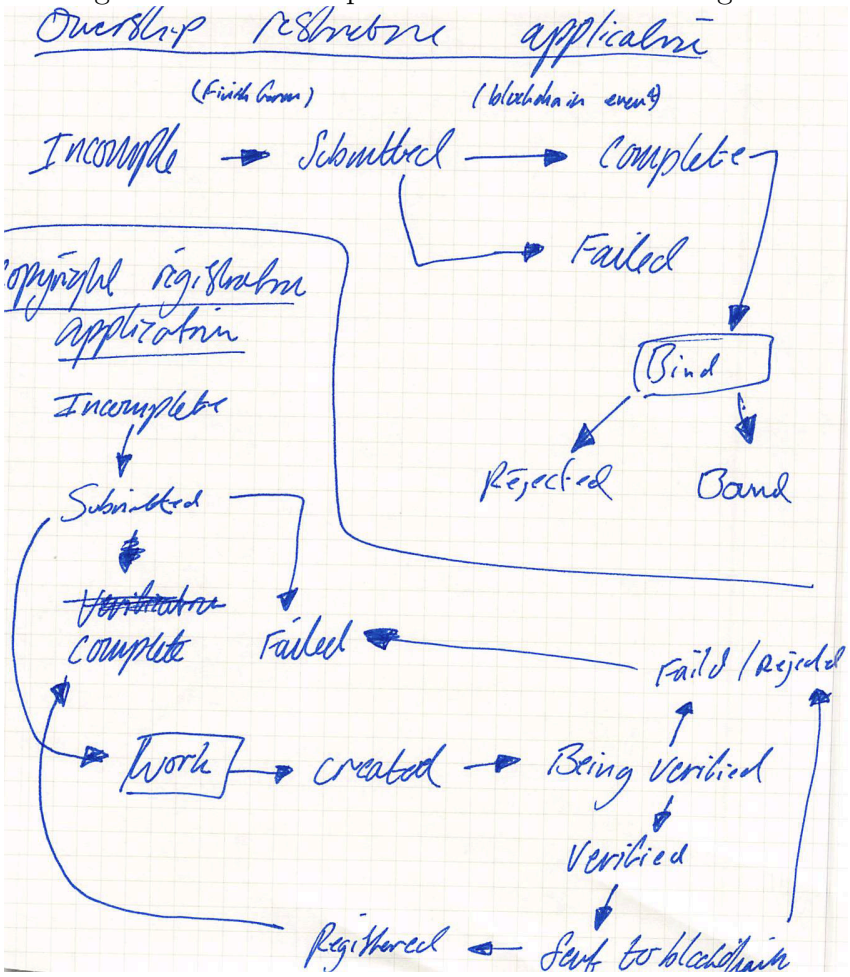


Figure 72: Original dispute handling flow diagram

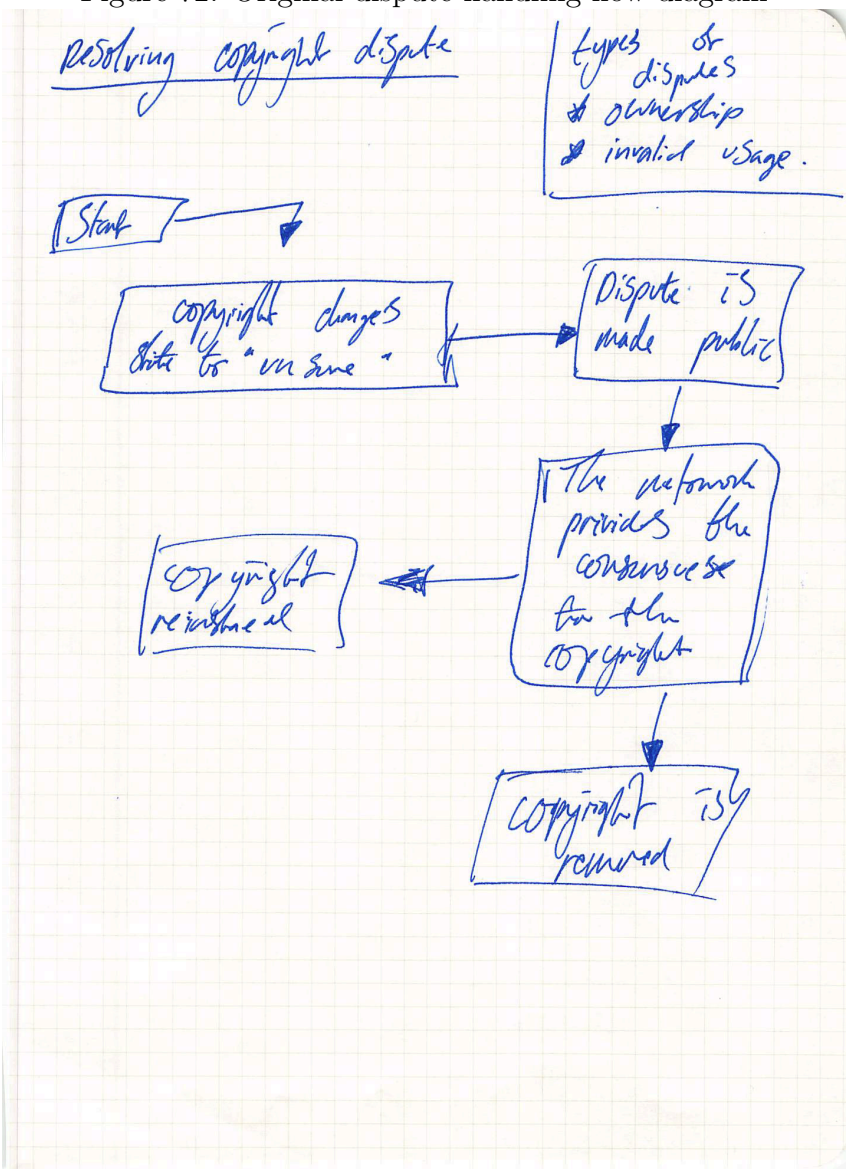
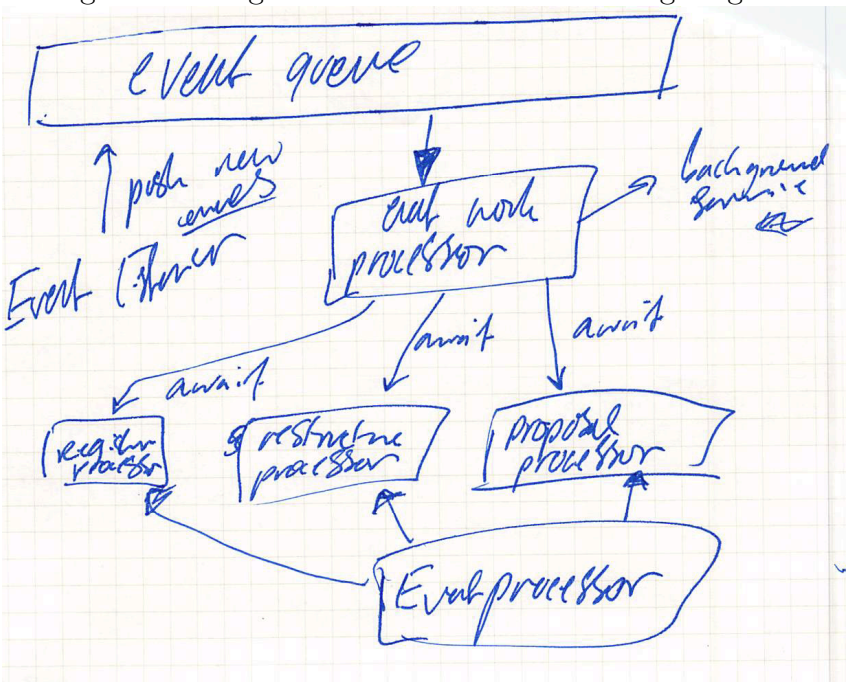


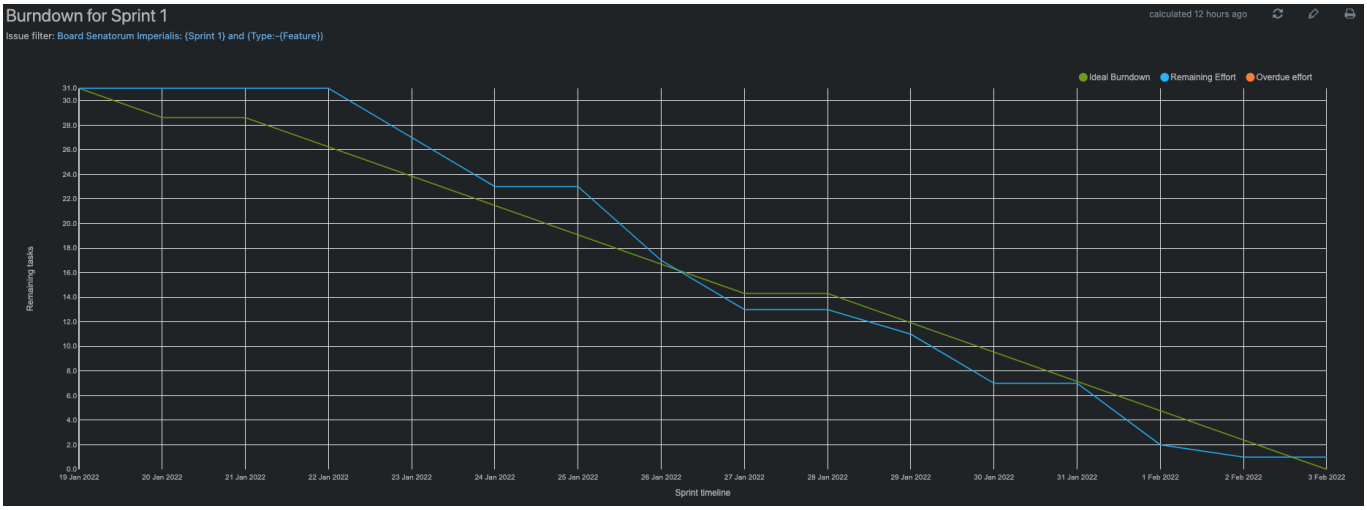
Figure 73: Original blockchain event listening diagram



10.4 Sprint reviews

Sprint reviews can also be found on the README.

Sprint 1 (19th Jan – 2nd Feb)



Focus

- Smart contracts
- User accounts
- Authentication
- File hashing
- File signing

What was achieved

All tasks and functionality assigned to this sprint have been completed. A Copyright interface was written drawing inspiration from the EIP-721 contract interface built for non-fungible tokens, many of the core concepts of this token are applicable to copyright contracts and was a firm base to start from. However this contract doesn't support multi-party ownership which is core tenant of this system (a book can obviously have two authors), to solve this issue I built the IStructuredOwnership interface which outlines multi-party ownership of a " work" using a share based system lifted from limited companies.

Then an abstract implementation of these interfaces was built simply called Copyright this was then combined with an ICopyrightMeta interface to create derived contracts representing three types of copyright licenses found here

A user account model was developed and migrated into a MySQL database representing a user in the system, to authenticate these users I used MetaMask to get a users wallet address sign a message with a randomly generated nonce which then gets decoded on the server to verify the ownership of the original wallet address. All this results in a quick and easy one click login flow for the system simplifying authentication for user and the system itself.

File hashing is handled by a basic sha-512 implementation offered in .NET, the user uploaded file is also signed both by a number of individual WorkSigners specially built for different content types (image, sound, video, text/pdf) and a universal signer that encodes a digital signature into the file.

What was not achieved

Expansive unit testing of Angular components and services was not completed only basic sanity checks, this is justified as frontend flow is subject to change based on user feedback and when the frontend if fully fleshed out (only simple login, logout buttons, file upload and user info wizard components were built this sprint).

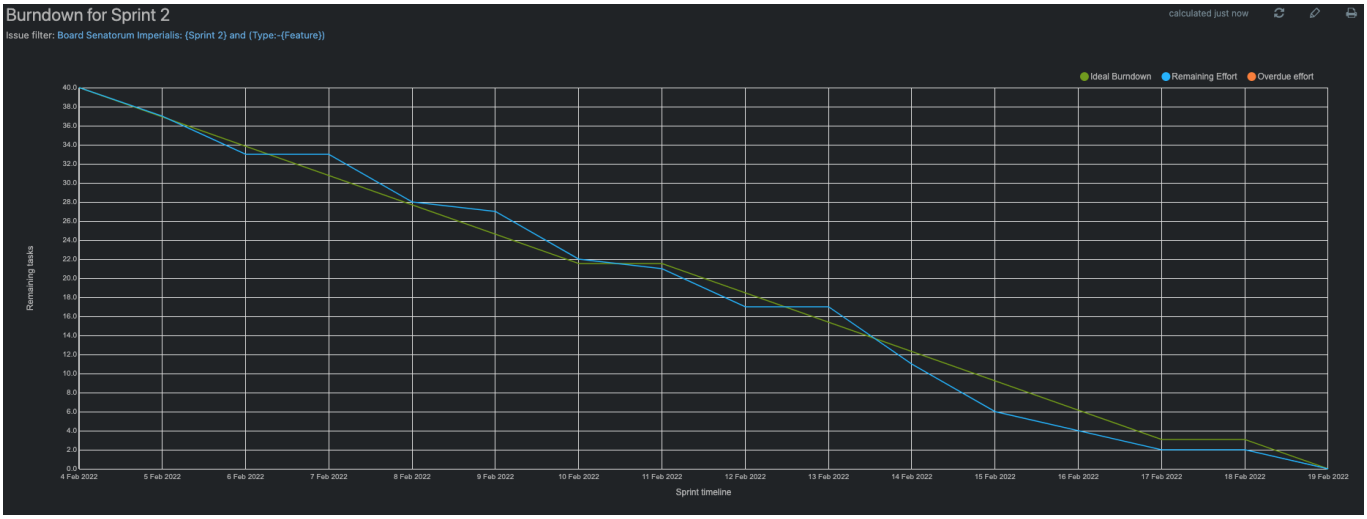
What went wrong

Proper forethought into user authentication and the interplay between Ethereum accounts and accounts track on the system was not taken which resulted in more development time wasted figuring this logic out.

Conclusion

This first sprint was very successful achieving all goals set out, smart contracts are in a good place, users can login, fill out personal info, upload work and download a digitally signed master file.

Sprint 2 (4th Feb – 18nd Feb)



Focus

- Copyright registration
- Restructure ownership
- Applications
- Multi-party proposal binding
- Search
- Blockchain data injection

What was achieved

All tasks and functionality assigned to this sprint have been completed excluding copyright un-registration which was removed from the sprint part way through over concerns that the feature is unnecessary and would need substantial thought and debate before implementing.

The first focus of this sprint was the "application" framework which resulted in the FormsService as generic interface for interacting and manipulating model driven forms (application), this code allows me to add new forms very easily with infrastructure already built. The working principle is leveraging the power of **OOP** to create a base Application class that all subsequent applications inherit (CopyrightRegistrationApplication)

these application models are then manipulated with Submitters and Updaters which contain any extra logic to be processed when an application is updated or submitted custom to that type of application. All these different applications are even stored in the same table on the database thanks to EFCores support for abstract classes which adds columns for all properties of all inheritors plus a discriminator column to represent the inherited class.

Once the application framework was built it was time to build specific applications and of course the first was the copyright registration application which used Angular reactive forms (cpy-registration-form) and the input model to update and submit the finished application.

After submitting these applications the system was going to need to communicate with the blockchain sending registration, propose and bind transactions. Using the BlockchainConnection from the first sprint and the generated contract service (thank you Nethereum) it was relatively easy to send these transactions but the processing of them in the real world can take time (as a result of the Ethereum blockchain) so I didn't want wait for the transaction to be proofed and mined all within the same http call. My solution for this was to setup event listeners which poll the blockchain for specific "Events" and then process them running all the necessary logic based on the type of event and payload data.

However once the copyrights are registered onto the blockchain I need a way of querying it to present relevant data to the user, this was simple as Nethereum generates all the necessary infrastructure and all I have to do is call the service methods, you can see this in the get works methods which use the injectFromChain method to get data from the chain.

Search has been implemented as an endpoint with all logic tested ready for the frontend ui to be built, I chose to focus all my ui and design efforts towards the applications and flow of registration and restructures therefore only the backend was deemed workable for this sprint.

What was not achieved

As stated above the unregister feature was dropped this sprint as its worth was called into question as currently theres no real way of removing a copyright registration in the legal world in practice a copyright is no longer enforced by a rights holder when the holder no longer actively pursues claims of infringement or the work moves into the public domain. Therefore giving more reason and though to expiry functions in my contract over a specific un register method.

What went wrong

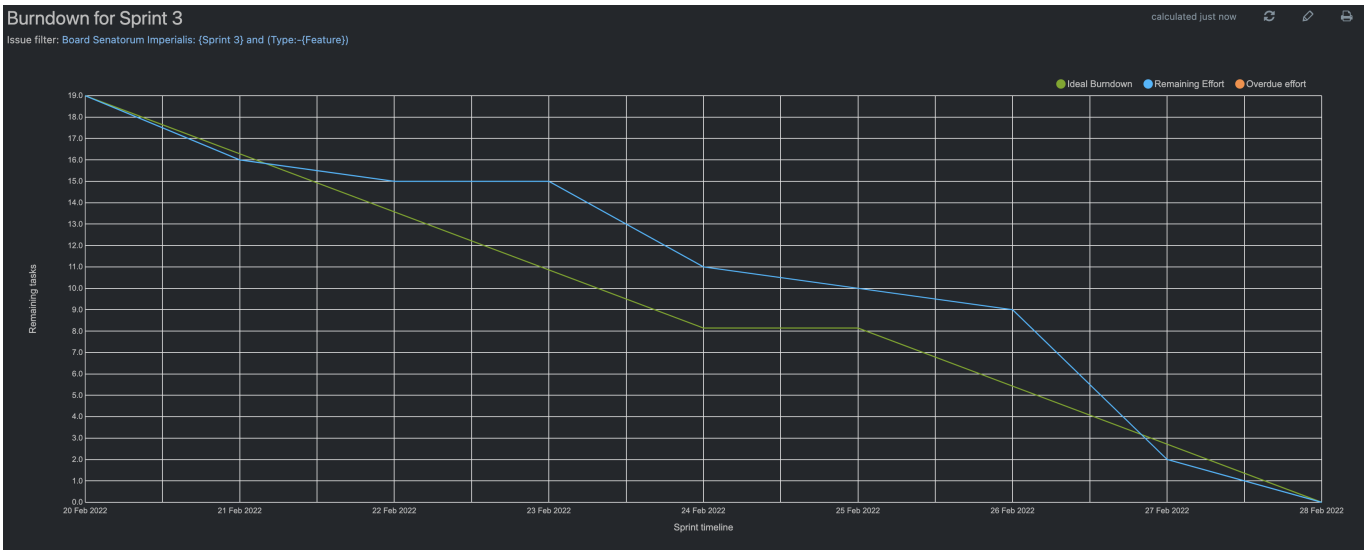
There was many tasks and features that conflicted with underlying design changes made in the first sprint, this is because all the work sanctioned for this sprint was backlog and chosen before any development work started. This is a failure on my part to properly review the upcoming work for this sprint after the last, when partaking in the sprint review I thought thoroughly on what was achieved but not the consequences that work had on the entire system and the work to be completed.

To mitigate this failure for the upcoming sprint all tasks and features already submitted will be used as a guide not as a list of what to do next hopefully producing a more accurate backlog of issues to complete.

Conclusion

This second sprint was very successful giving users the ability to register copyrights, restructure the ownership of those copyrights and injection of data stored on the blockchain.

Sprint 3 (20th Feb – 27th Feb)



Focus

- Dispute filing
- Dispute resolution
- User focused UI elements
- Copyright expiry
- ChainSync™
- Account delete
- Wallet transfer

What was achieved

All tasks and functionality assigned to this sprint have been completed.

The first and most important piece of functionality to be implemented for this sprint was to file copyright disputes, this was done using my existing applications framework with a DisputeApplication data model representing what will be needed to file a dispute. Because of the existing framework this side of the implementation was quick and most of the development time was spent on changing and improving the data model for the needs of the application. Next was to create a new dispute service for handling resolution of the dispute, a rights holder can then accept or reject the expected recourse described in the application. I've built two expected recourses: change of ownership that transfers the copyright to the accusing party and payment which transfers a set amount of Eth to the accuser stated in the application.

Next was handling expiry of copyrights, instead of an explicit state expiry happens via a modifer or check that happens on requests and transactions with the contract and throws an error if the right is expired I catch this error and set the work as expired silently in a background service and queue.

```
modifier isExpired(uint256 rightId)
{
```

```
        require(_metadata[rightId].expires > block.timestamp, EXPIRED);
        _;
    }
```

Now with a lot of state change and interaction with the chain I need a way of ensuring the system is in sync and consistent with the blockchain, to do this I created ChainSync which is made up of a background service that runs batches of synchronisation this synchronisation comes from any number of ISynchronisers and at this time an OwnershipSynchroniser has been implemented which checks the ownership structure on the chain and compares it with our database.

The final feature to implement was two account configuration methods that: transfer ownership and delete account. These again used the application framework and a new AccountManagementService .

What was not achieved

All work assigned was completed however I am not confident all edge cases have been covered therefore the reliability of the code could be questioned.

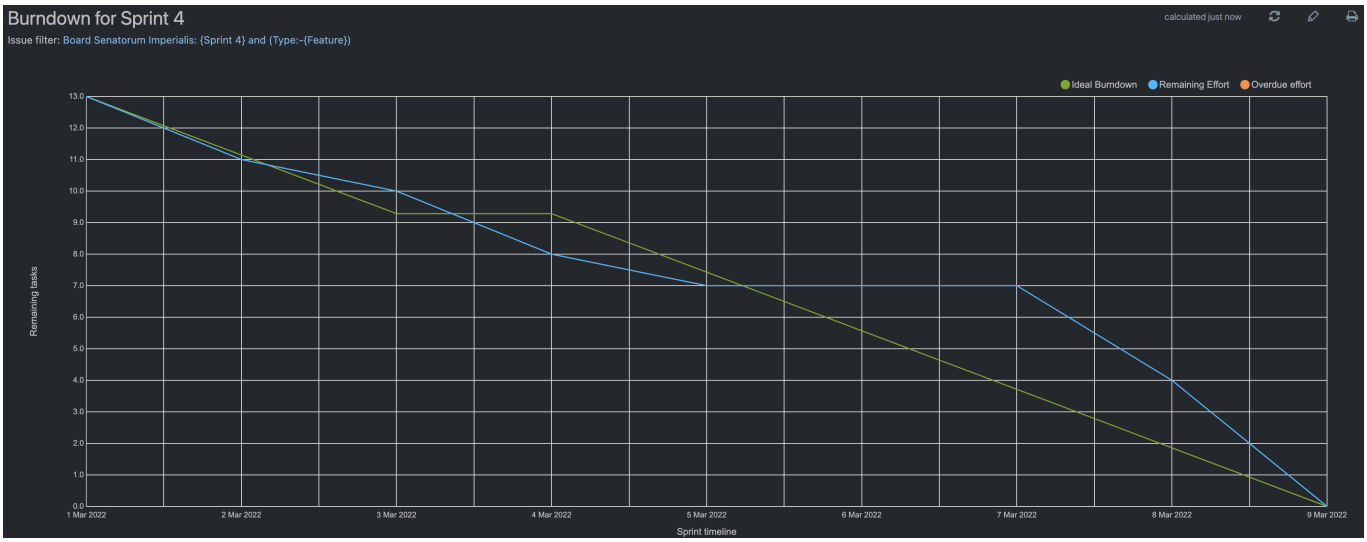
What went wrong

When building out the dispute filing and resolution system the design and operation of the system was not throughout so significant time was spent trying different paths.

Conclusion

This sprint was very successful dispute a lot of work to be done in only a week, it gave the user the ability to file and resolve disputes, delete account and transfer assets to wallet. The system now recognises expired copyrights and keeps in sync with the blockchain.

Sprint 4 (1st March – 8th March)



Focus

- Bug fixes
- Work CDN & Usage (Extra feature)

- Websocket
- Continuous deployment

What was achieved

All tasks an functionality assigned to this was implemented in the time period.

This sprint was focused first on two extra features I wanted to get implemented first was a decentralised blockchain based content delivery network, after researching a number of solutions one stood out as the best option being IPFS (InterPlanetary File System) which is a peer to peer hypermedia protocol to retrieve files saved across multiple nodes/peers in chunks. So I installed Ipfs and started up my own node needed to interact and most importantly add files to the network, using the Ipfs.Http.Client library it was then easy to add the digitally signed to this network returning a CID which is a unique identifier (essentially a hash of the file) used to retrieve the file from the Ipfs network. This identifier is saved on the work.

Once saved to the network a link to the work is created but I wanted a way for creators to measure the number of times their work has been used, I decided on creating a reverse proxy which takes a request for a work records the use and then passes the ipfs request onto the client.

The second extra feature implemented was websocket integration which allows for realtime updates to the client which in a blockchain based application sending transactions that can possibly take any amount of time to complete greatly improves the usability of the application. This was implemented using register and push methods, when a user loads an application the client will send a listen command to the backend, whenever an application or work is updated those updates will pushed out to listeners.

What was not achieved

Although all bugs registered in this sprint were fixed a level of polish was and has not been obtained, everything technically works but the software is still not "user proof"

What went wrong

Work on implementing the websocket radically changed a large portion of the frontend dataflow to accommodate the now realtime asynchronous data that can now change on the fly at anytime and therefore has to change and adapt. This caused more problems than expected and therefore implementing this feature took considerably more time and change.

Conclusion

This sprint was very successful implementing extra quality of life features, the project now feels like a more complete and holistic product ready to be used.

10.5 User guide

The user guide can be found at https://github.com/MrHarrisonBarker/CRPL/wiki

If you want to run this software locally on your machine then I would strongly suggest following my instructions on the README for installation and running.

10.6 Test results

10.6.1 Smart contract unit test results

```
ApproveManager
  ✓ Should approve manager for user

ApproveOne
  ✓ Should approve for right
  ✓ Should REVERT when not valid rightId
  ✓ Should REVERT when not valid address
  ✓ Should REVERT when not sent from shareholder or approved

ProposeRestructure
  ✓ Should REVERT when expired

GetApproved
  ✓ Should registerer be approved
  ✓ Should REVERT when not valid rightId

IsManager
  ✓ Should be manager

PortfolioSize
  ✓ Should have portfolio
  ✓ Should REVERT when not valid address

Register
  ✓ Should register new copyright
  ✓ Should REVERT no shareholders
  ✓ Should REVERT when invalid shareholders

Meta
  ✓ Should register with metadata
  ✓ Should get registered metadata
  ✓ Should REVERT no shareholders
  ✓ Should REVERT when invalid shareholders
```

```
MultiOwner
  ✓ Should have ownership structure with many addresses
  ✓ Should fail to bind proposal
  ✓ Should not bind proposal if one vote

SingleOwner
  ✓ Should approve for right
  ✓ Should have ownership structure of one address
  ✓ Should propose new structure
  ✓ Should bind a restructure

BindRestructure
  ✓ Should bind a restructure
  ✓ Should REVERT when not sent from shareholder or approved
  ✓ Should REVERT when not valid rightId
  ✓ Should REVERT when not valid rightId

CurrentVotes
  ✓ Should get no current votes
  ✓ Should get current votes
  ✓ Should REVERT when not valid rightId

OwnershipOf
  ✓ Should have ownership structure
  ✓ Should REVERT when not valid rightId

Proposal
  ✓ Should have no proposal
  ✓ Should REVERT when not valid rightId

ProposeRestructure
  ✓ Should propose new structure
  ✓ Should REVERT when not valid rightId
  ✓ Should REVERT no shareholders
  ✓ Should REVERT when invalid shareholders
  ✓ Should REVERT when not sent from shareholder or approved
```

10.6.2 Back-end unit test results

Test run details

Total tests	Passed : 163 Failed : 0 Skipped : 2	Pass percentage	Run duration
165		98 %	17s 916ms

All Results

/Users/harrison/Desktop/CRPL/CRPL.Tests/bin/Debug/net6.0/CRPL.Tests.dll

✔ Should_Submit_And_Start_Registration	1s 745ms
✔ Should_Submit_And_Delete	48ms
✔ Should_Submit	103ms
✔ Should_Submit_And_Propose	50ms
✔ Should_Submit_And_Transfer	28ms
✔ Should_Assign_Owners	25ms
✔ Should_Throw_If_No_Shareholder	50ms
✔ Should_Update	57ms
✔ Should_Update_And_Encode_Ownership_Stakes	19ms
✔ Should_Update	21ms
✔ Should_Assign_User_To_Application	70ms
✔ Should_Attach_Work_To_Application	53ms
✔ Should_Update	21ms
✔ Should_Assign_Users	20ms
✔ Should_Throw_If_No_Shareholder	23ms
✔ Should_Update	20ms
✔ Should_Throw_When_Address_Not_Valid	22ms
✔ Should_Update_And_Assign	21ms
✔ Account_Should_Have_Balance	154ms
✔ Should_Have_Account	8ms
✔ Should_Get_Ownership	4s 67ms
✔ Should_Register_New_Copyright	3s 976ms
✔ Should_Set_Status	153ms
✔ Should_Throw_If_No_Application	30ms
✔ Should_Throw_If_No_Application	106ms
✔ Should_Update_Status	27ms
✔ Should_Sign	58ms
✔ Should_Throw_If_No_Work	22ms
✔ Should_Update_Application_Status	41ms
✔ Should_Update_Work_Status	28ms
✔ Should_Assign_Shareholders	135ms
✔ Should_Set_Origin_To_Complete	47ms
✔ Should_Set_Status	31ms
✔ Should_Throw_If_No_Application	25ms

✔ Should_Throw_If_No_User 25ms
✔ Should_Throw_If_No_Work 22ms
❢ Should_Get 1ms
Error:

```
OneTimeSetUp:
```

✔ Should_Add 128ms
✔ Map_Should_HaveValidConfig 123ms
✔ Should_Migrate_And_Seed 30ms
✔ Should_Propose_When_Multi_Ownership 144ms
✔ Should_Propose 66ms
✔ Should_Attach_Work_To_Application 80ms
✔ Should_Not_Add_If_Already_Assigned 38ms
✔ Should_Throw_If_No_Work 28ms
✔ Should_Throw_If_Work_Not_Registered 27ms
✔ Should_Send_Transaction_With_Application 43ms
✔ Should_Send_Transaction_With_Work 43ms
✔ Should_Throw_If_No_Application 26ms
✔ Should_Throw_If_No_Work 30ms
✔ Should_Send_To_Expired_Queue 46ms
✔ Should_Send_Transaction 30ms
✔ Should_Throw_If_No_Work 29ms
✔ Should_Throw_When_No_Dispute 51ms
✔ Should_Update_Result 49ms
✔ Should_Record_Payment 34ms
✔ Should_Throw_When_No_Dispute 24ms
✔ Should_Throw_When_No_Dispute 28ms
✔ Should_Update_Result 37ms
✔ Should_Create_And_Submit_Restructure 37ms
✔ Should_Throw_When_No_Dispute 25ms
✔ Should_Throw_When_No_Work 26ms
✔ Should_Throw_When_Not_ChangeOfOwnership 28ms
✔ Should_Throw_When_Not_Submitted 28ms
✔ Should_Cancel 48ms
✔ Should_Throw_When_Not_Found 21ms
✔ Should_Map_To_Delete 58ms
✔ Should_Map_To_Dispute 32ms
✔ Should_Map_To_Ownership 28ms
✔ Should_Map_To_Registration 32ms
✔ Should_Map_To_Wallet_Transfer 35ms
✔ Should_Get_Users_Applications 101ms
✔ Should_Be_Submitted 28ms
✔ Should_Throw_If_Already_Complete 16ms
✔ Should_Throw_If_Already_Submitted 17ms
✔ Should_Throw_If_Not_Found 16ms

✔ Should_Create_New_And_Save 32ms
✔ Should_Create_New_If_Not_Found 27ms
✔ Should_Save_Updates 38ms
✔ Should_Get_All 67ms
✔ Should_Get_All 47ms
✔ Should_Get_Users_Works 67ms
❢ Should_Get_Work < 1ms
Error:

```
overflow error
```

✔ Should_Send_To_Expired_Queue 54ms
✔ Should_Be_Type 52ms
✔ Should_Only_Take_One 41ms
✔ Should_Search_Keyword 48ms
✔ Should_Search_Sort_By 204ms
✔ Should_Complete 40ms
✔ Should_Not_Set_Failed_If_Wrong_Throw 20ms
✔ Should_Set_Status_Failed_When_Throw 27ms
✔ Should_Set_Status_To_Sent 23ms
✔ Should_Throw_If_No_Application 17ms
✔ Should_Throw_If_No_Associated_Work 18ms
✔ Should_Start_Registration 33ms
✔ Should_Listen 10ms
✔ Should_Listen_To_Existing 4ms
✔ Should_Listen 5ms
✔ Should_Listen_To_Existing 4ms
✔ Should_Register 5ms
✔ Should_Register_To_Existing 4ms
✔ Should_Remove 14ms
✔ Should_At_Least_One_Be_Not_Real 41ms
✔ Should_Be_Real 16ms
✔ Should_Many_Be_Real 17ms
✔ Should_Not_Ne_Real 17ms
✔ Should_Assign_Using_Address 31ms
✔ Should_Throw_If_No_User 17ms
✔ Should_Assign 29ms
✔ Should_Throw_If_No_User 16ms
✔ Should_Authenticate 25ms
✔ Should_Throw 17ms
✔ Should_Authenticate_For_30_Days 157ms
✔ Should_Be_Invalid 17ms
✔ Should_Generate_Authentication_Token 140ms
✔ Should_Regenerate_Nonce 32ms
✔ Should_Return_User_If_Authenticated 26ms
✔ Should_Throw_Not_Found 17ms

✔ Should_Fetch_New	23ms
✔ Should_Save_New_Account	22ms
✔ Should_Save_Nonce	22ms
✔ Should_Get_Account	26ms
✔ Should_Get_Partial_Fields_If_Not_Complete	19ms
✔ Should_Throw_Not_Found	16ms
✔ Should_Be_Share_Holder	33ms
✔ Should_Not_Be_Share_Holder	17ms
✔ Should_Throw_If_No_User	17ms
✔ Should_Be_Unique	23ms
✔ Should_Not_Be_Unique	15ms
✔ Should_Be_Unique	18ms
✔ Should_Not_Be_Unique	16ms
✔ Should_Revoke	24ms
✔ Should_Throw	17ms
✔ Should_Return_Partials	26ms
✔ Should_Throw_Not_Found	17ms
✔ Should_Update_Status_When_Complete	24ms
✔ Should_Update_UserAccount	17ms
✔ Should_Add_File_To_Ipfs	26ms
✔ Should_Remove_Cached_Original_From_Repo	13ms
✔ Should_Return_Signed_Work	11ms
✔ Should_Throw_When_Not_Registered	11ms
✔ Should_Already_Exist	21ms
✔ Should_Have_Content	12ms
✔ Should_Upload	11ms
✔ Should_Be_Authentic	52ms
✔ Should_Not_Be_Authentic	20ms
✔ Should_Throw_If_No_Application	< 1ms
✔ Should_Throw_If_No_Work	13ms
✔ Should_Be_Different	56ms
✔ Should_Be_The_Same	13ms
✔ Should_Be_Different	30ms
✔ Should_Be_The_Same	11ms
✔ Should_Catch_Expired_Work	13ms
✔ Should_Throw_If_No_Work	11ms
✔ Should_Proxy	172ms
✔ Should_Register_Usage	144ms
✔ Should_Throw_If_Id_Not_Found	16ms
✔ Should_Throw_If_No_Work	16ms
✔ Should_Throw_If_Not_Guid	18ms
✔ Should_Add_Metadata_To_JPG	592ms
✔ Should_Add_Metadata_To_MP3	138ms
✔ Should_Add_Metadata_To_WAV	43ms
✔ Should_Add_Metadata_To_PDF	200ms

✔ Should_Add_Signature 9ms
✔ Should_Add_Metadata_To_MP4 75ms

Informational messages

NUnit Adapter 4.0.0.0: Test execution started
Running all tests in /Users/harrison/Desktop/CRPL/CRPL.Tests/bin/Debug/net6.0/CRPL.Tests.dll
NUnit3TestExecutor discovered 165 of 165 NUnit test cases using Current Discovery mode, Non-Explicit run
The test account balance is 10880332376529842976277890 which is 10880332.37652984297627789 eth
There are 2 accounts
Status -> 1
Should_Get: OneTimeSetUp:
Should_Get_Work: overflow error
NUnit Adapter 4.0.0.0: Test execution complete

10.6.3 Front-end unit test results

Karma v 6.3.17 - connected; test: complete;

DEBUG

Chrome 100.0.4896.88 (Mac OS 10.15.7) is idle

Jasmine 3.8.0

Options

35 specs, 0 failures, randomized with seed 25763 finished in 0.137s

FormsService
- should be created

ValidatorsService
- should add null when validating email
- should add error when validating email
- should be created
- should add error when validating phone
- should add null when validating phone

AlertService
- should return alert subject
- should be created
- SPEC HAS NO EXPECTATIONS should alert
- should set loading false on construct
- should start loading
- should stop loading

QueryService
- should be created

ResonanceService
- should be created

ExternalService
- should be created

WarehouseService
- should be created

AuthService
- should fetch nonce
- should logout
- should use exist user
- should authenticate
- should be created

WorksService
- should get signed work
- should be created

CopyrightService
- should be created

AuthGuard
- should route away if no token
- should return true if already authenticated
- should be created
- should return true if authenticated
- should route away if error thrown

RememberedGuard
- should be created

UserService
- should be created
- should update account
- should throw if not authenticated
- should find if phone exits
- should find if email exits

References

[1] Berne convention for the protection of literary and artistic works. Available at: https://treaties.un.org/pages/showdetails.aspx?objid=0800000280115ec9.

[2] Bitcoin: A peer-to-peer electronic cash system. Available at: https://www.ussc.gov/sites/default/files/pdf/training/annual-national-training-seminar/2018/Emerging_Tech_Bitcoin_Crypto.pdf.

[3] Copyright:rights granted. Available at: https://en.wikipedia.org/wiki/Copyright#Rights_granted.

[4] Ethereum improvement proposals. Available at: https://eips.ethereum.org.

[5] Manifesto for agile software development. Available at: https://agilemanifesto.org.

[6] Proof-of-stake. Available at: https://ethereum.org/en/developers/docs/consensus-mechanisms/pos.

[7] Proof-of-work. Available at: https://ethereum.org/en/developers/docs/consensus-mechanisms/pow.

[8] Tweet complaining about dmca takedown abuse gets hit with dmca takedown. Available at: https://www.theverge.com/2019/4/15/18311091/piracy-tweet-dmca-takedown-request-starz-eff-american-gods.

[9] Wipo copyright treaty. Available at: https://treaties.un.org/Pages/showDetails.aspx?objid=08000002800838a5&clang=_en.

[10] Balázs Bodó, Daniel Gervais, and João Pedro Quintais. Blockchain and smart contracts: the missing link in copyright licensing? *International Journal of Law and Information Technology*, 26(4):311–336, 09 2018. Available at: https://doi.org/10.1093/ijlit/eay014.

[11] Rainer Bohme, Nicolas Christin, Benjamin Edelman, and Tyler Moore. Bitcoin: Economics, technology, and governance. *Journal of economic Perspectives*, 29(2):213–38, 2015. Available at: https://www.aeaweb.org/articles?id=10.1257/jep.29.2.213.

[12] Aggelos Kiayias and Giorgos Panagiotakos. Speed-security tradeoffs in blockchain protocols. *IACR Cryptol. ePrint Arch.*, 2015:1019, 2015. Available at: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.737.9945&rep=rep1&type=pdf.

[13] Christopher Millard. Blockchain and law: Incompatible codes? *Computer Law & Security Review*, 34(4):843–846, 2018. Available at: https://www.sciencedirect.com/science/article/pii/S0267364918302437.

[14] Satoshi Nakamoto. Bitcoin whitepaper. 2008. Available at: https://bitcoin.org/bitcoin.

[15] Paul Rimba, An Binh Tran, Ingo Weber, Mark Staples, Alexander Ponomarev, and Xiwei Xu. Comparing blockchain and cloud services for business process execution. pages 257–260. IEEE, 2017. Available at: https://ieeexplore.ieee.org/abstract/document/7930226.

[16] Alexander Savelyev. Copyright in the blockchain era: Promises and challenges. *Computer Law & Security Review*, 34(3):550–561, 2018. Available at: https://www.sciencedirect.com/science/article/pii/S0267364917303783.

[17] Shihab Shahriar Hazari and Qusay H. Mahmoud. Improving transaction speed and scalability of blockchain systems via parallel proof of work. *Future Internet*, 12(8), 2020. Available at: https://www.mdpi.com/1999-5903/12/8/125.

[18] Alan T. Sherman, Farid Javani, Haibin Zhang, and Enis Golaszewski. On the origins and variations of blockchain technologies. *IEEE Security Privacy*, 17(1):72–77, 2019. Available at: https://ieeexplore.ieee.org/abstract/document/8674176.

[19] Bayu Adhi Tama, Bruno Joachim Kweka, Youngho Park, and Kyung-Hyune Rhee. A critical review of blockchain and its current applications. In *2017 International Conference on Electrical Engineering and Computer Science (ICECOS)*, pages 109–113, 2017. Available at: https://ieeexplore.ieee.org/abstract/document/8167115.

[20] D. Tapscott and A. Tapscott. *Blockchain Revolution: How the Technology Behind Bitcoin and Other Cryptocurrencies Is Changing the World*. Penguin Publishing Group, 2018. Available at: https://books.google.co.uk/books?id=8qlPEAAAQBAJ.

[21] Laura Wood. Global blockchain market report 2021: Market size is projected to grow from 4.9*billionin*2021*to*67.4 billion by 2026, at a cagr of 68.4 Available at: https://finance.yahoo.com/news/global-blockchain-market-report-2021-121300653.html.

[22] Svein Ølnes, Jolien Ubacht, and Marijn Janssen. Blockchain in government: Benefits and implications of distributed ledger technology for information sharing. *Government Information Quarterly*, 34(3):355–364, 2017. Available at: https://doi.org/10.1016/j.giq.2017.09.007.