Entrades NFT per a esdeveniments: Implementació de transaccions segures mitjançant Smart Contracts de Ethereum

Treball Final de Grau 2024/25

Informe de Progrés I

UAB
Universitat Autònoma de Barcelona

Lucas Dalmau Garcés – 1636290

ÌNDEX

Metodologia aplicada

Marc de treball pel desenvolupament del Smart Contract

Primerament, i com a inici per començar el desenvolupament, s'ha realitzat una recerca exhaustiva per identificar eines que em permetin realitzar el desenvolupament d'aquesta primera part. Però, abans, s'han definit alguns requisits que hauria de tenir l'eina a banda de poder compilar el Smart Contract:

- L'eina ha de permetre crear una xarxa de proves d'Ethereum, de forma que pugui fer desplegaments en local.
- Ha d'incloure un sistema per a la implantació de tests automàtics per així poder testejar els casos límit i frontera.
- En últim lloc, i més com a preferència, seria de gran valor que es pugui integrar amb Visual Studio Code, estigui basada en un llenguatge que domini per agilitzar l'aprenentatge de l'eina i compti amb una documentació exhaustiva que faciliti una corba d'aprenentatge gradual, de forma que pugui centrar tots els recursos en l'aprenentatge de Solidity i el desenvolupament del Smart Contract.

Una vegada definits, tal com es va esmentar dintre de l'Informe Inicial, a continuació es presenta una anàlisi dels candidats:

Hardhat

Tal com se cita dintre de la documentació, aquesta eina és un "entorn de desenvolupament de Software per a Ethereum". A més, està basat en Node.js el qual va ser un dels Frameworks que s'han estudiat durant el curs de "Sistemes i Tecnologies Web". A l'estar basat en Node.js tenim el desavantatge que és monofil, fet que implica que tots els esdeveniments es processen dintre del mateix fil d'execució desaprofitant la capacitat de processadors moderns, posicionant a aquesta opció com una que en termes de rendiment és inferior a altres. No obstant això, aquesta eina presenta dues característiques rellevants pel projecte.

En primer lloc, la Hardhat Network, aquesta funcionalitat permet la creació d'una xarxa local d'Ethereum on es configura automàticament un node que es connecta a aquesta xarxa local per poder desplegar els contractes, executar-los i poder-los testejar. A més aquesta xarxa obre un port dintre de la màquina de forma que es pot connectar amb altres clients i wallets com per exemple Metamask, de forma que possibilita fer simulacions reals de la interacció entre el wallet i la xarxa local. Això aporta moltíssim al projecte perquè des del primer moment puc simular un entorn real.

D'altra banda, implementa un sistema per a poder fer tests automàtics amb JavaScript, de forma que, a mesura que es va desenvolupant el Smart Contract, es pot desenvolupar tests per avaluar els casos límit i frontera a fi de poder depurar el comportament.

En últim lloc, he trobat de molt valor que totes les funcionalitats estan molt detallades amb exemples pas a pas per poder practicar, de forma que l'aprenentatge és més guiat i gradual per a després aplicar-ho al mateix treball.

Foundry

Aquesta opció també és una eina pel desenvolupament de Smart Contracts, però amb la diferència d'estar basada en Rust. Això permet que l'execució sigui multifil, tenint una diferència en rendiment amb altres opcions com la que hem vist abans. També permet la creació d'una xarxa local d'Ethereum i es poden dur a terme tests amb la particularitat de què s'escriuen en el mateix llenguatge que el Smart Contract, eliminat així la necessitat d'utilitzar un llenguatge addicional. Cal mencionar que Foundry és una eina més recent i es denota en la documentació, ja que no està tan detallada i encara que aporta algun exemple per poder entendre-la pot arribar a dificultar l'autoaprenentatge, derivant més temps en aprendre a fer-la servir.

Remix IDE

Durant la recerca, he pogut observar que és una eina molt popular pel desenvolupament de Smart Contracts, té una documentació molt àmplia i ben construïda. Remix permet igual que les altres dues opcions tenir una xarxa "local" (ja que és un IDE en línia) per poder desplegar i testejar el Smart Contract i les proves automatitzades s'escriuen en Solidity. A més, al moment de desplegar un Smart Contract a la seva xarxa local, genera automàticament una interfície senzilla per poder provar manualment les funcions del Smart Contract. Indagant una mica més, he descobert que, encara que és una eina web, pot connectar-se a xarxes locals d'Ethereum creades amb altres eines com les vistes anteriorment. Possibilitant connectar wallets com el de Metamask per poder provar el funcionament del Smart Contrcat fent-ne ús de la interfície que genera. Aquesta part m'ha semblat molt interessant perquè em permet combinar una de les eines anteriors amb aquesta per "simular" el comportament de la Dapp amb el Smart Contract i el wallet. És per això que encara que no sigui el IDE de preferència, podria anar bé per testejar el comportament del Smart Contract de forma més pròxima al resultat final d'aquest projecte sense encara haver començat la segona part del desenvolupament.

Selecció final

Finalment, després d'analitzats els tres candidats, s'ha observat com la diferència entre Hardhat i Foundry és mínima, i s'ha arribat a la concussió de què ambdues eines son igual d'útils. Si bé és veritat que, tècnicament Foundry pot ser més ràpid en temps de compilació i execució de tests. Per les característiques d'aquest projecte, no considero com rellevant una execució que estalvií mil·lisegons en testeig.

D'altra banda, Remix ja és un IDE complet amb tot el necessari per poder desenvolupar un Smart Contract, però tal com vaig esmentar a l'Informe Inicial, prefereixo poder utilitzar un IDE que ja coneix com Visual Studio Code.

És així com davant de la decisió entre fer servir Hardhat o Foundry, veient que en funcionalitats són pràcticament idèntiques i satisfan els meus principals requisits, he decidit basar la decisió en un factor de preferència com és la corba d'aprenentatge. Durant aquesta avaluació d'ambdues eines, he pogut analitzar la seva documentació i el nivell d'informació que hi ha sobre ambdues eines, i Foundry al ser més nou, els exemples i els recursos per aprendre de forma autònoma em semblen més limitats en comparació amb Hardhat. A més, Hardhat té com a base tecnologies que conec àmpliament com Node.js que poden fer més dinàmic el desenvolupament de tests. És

per això que, com a eina final per desenvolupar aquest projecte he decidit utilitzar Hardhat dintre de Visual Studio Code.

Aprenentatge: Solidity i Smart Contracts

Abans de començar el desenvolupament del Smart Contract, s'ha agut d'aprendre el llenguatge que utilitza: Solidity. Per aquesta tasca, primerament he començat amb el llibre "*Mastering Ethereum : Building Smart Contracts and DApps*" i la documentació de Solidity, on he aprés la metodologia i algunes bones pràctiques com el patró de desenvolupament Checks-Effects-Interactions, amb el qual s'ha bast tot el desenvolupament del Smart contract. Per exemple aquest patró indica que primer s'han de fer les comprovacions pertinents abans d'executar res, per exemple, si la persona que invoca la funció està autoritzada, si ha enviat els fons suficients en cas de que sigui necessari, si es el propietari de la entrada, etc. Després, passa a la fase de "Effects" en la qual es modifiquen les variables d'estat que hi hagin dintre del smart contract, per exemple, en una venda, podria ser un canvi de la adreça propietària del NFT i per últim, les interaccions com podria ser retornar ethers si per exemple un usuari ha possat mes ethers dels que calien per fer una compra. Aquest patró seria molt important per a prevenir alguns atacs com el de reentrada que es comenta mes endavant.

Amb tota aquesta informació i la necessitat de posar en pràctica els coneixements per consolidar-los, he estat practicant amb l'eina de Hardhat desenvolupant petits Smart Contracts fixant-me en cadascun en un aspecte concret del desenvolupament amb Solidity, per a fer-ho, he basat aquesta part del aprenentatge en la web de "*Solidity by Example*" la qual conté Smart Contracts, dels quals he recreat els primers 44 exemples amb els coneixements que he anat adquirint. Aquesta ultima part ha sigut d'especial utilitat ja que mentre practico, he pogut familiaritzar-me mes amb l'eina de Hardhat, així com amb la creació de tests automàtics.

Finalment, i per aprendre una mica mes sobre com enfortir la seguretat del Smart contract, la pagina de "*Solidity by Example"* proporciona un apartat de "Hacks" amb el qual he pogut comprendre quins son els principals vectors d'atacs dintre dels Smart contracts i quines mesures son necessàries dintre del marc de treball proposat.

Avenços I: Desenvolupament del Smart Contract

Els avenços dintre d'aquesta primera part han sigut molt satisfactoris, ja que s'ha aconseguit el desenvolupament total del Smart Contract. En aquesta part es clarifiquen alguns aspectes del disseny així com s'amplia informació sobre les variables utilitzades i quins imports s'han fet servir juntament amb les respectives clarificacions.

Variables globals definides

TicketStatus Un tiquet pot tenir diferents estats segons que es vol fer amb ell. Per aquest treball s'han definit 3 estats *(Vegeu: "Concepte d'estat dels tiquets").*

- Active: És l'estat inicial d'un tiquet quan es compra, dins d'aquest estat es pot fer qualsevol acció amb ell.
- Pending: En aquest estat l'usuari pot presentar al validador (sigui una persona o sigui un control d'accés automatitzat) l'entrada a fi de poder bescanviar-la per

l'accés a l'esdeveniment. Aquest estat té una durada de 10 minuts, a partir de quan l'usuari fa la transacció pel canvi d'estat. Durant aquest, l'entrada no es pot transferir ni vendre.

- Redeemed: Estat final d'una entrada que ha estat bescanviada. Quan una entrada es troba en aquest estat, no pot ser posada en venda mitjançant els mètodes implementats, però sí que es pot transferir mitjançant la funció transfer() de l'estàndard ERC721 (*Vegeu: "Concepte de transferència d'una entrada"*).

Struct Ticket: És una estructura dissenyada per guardar informació relativa al tiquet, on els seus components són:

- status: L'estat actual del tiquet.
- pendingSince: Guarda la data del bloc en la qual el propietari de l'entrada ha canviat l'estat a pending, d'aquesta forma es pot comptar si el temps dels 10 minuts que dura aquest estat ha passat.
- salePrice: En cas que l'usuari posi a la venda el seu tiquet, en aquesta variable es defineix el preu de venda.
- seller: Variable on es guarda l'adreça del propietari que posa a la venda l'entrada *(Vegeu "Concepte de venda").*

(OBS: aquesta estructura no inclou el propietari, ja que d'això s'encarrega l'estàndard ERC721).

ticketPrice: Variable que s'inicialitza al constructor. Indica el preu per entrada que ha establert l'organitzador per les entrades en estoc.

maxSupply: Variable que s'inicialitza al constructor. Indica el nombre total d'entrades que es poden generar.

baseTokenURI: Adreça base per guardar metadades de l'esdeveniment relacionades amb el tiquet adquirit per l'usuari. *(OBS: s'ha realitzat prou implementació dintre del Smart Contract i s'avaluarà la seva implementació final al següent informe).*

maxTicketsPerAddress: Indica el nombre de tiquets que pot tenir/haver tingut una determinada adreça.

useTimeLimit: Booleà que indica si l'esdeveniment té data límit de forma que una vegada passat el temps ja no es permeten les compres ni les vendes d'entrades amb les funcions desenvolupades.

eventEndTime: Quantitat de temps especificat al constructor en hores sobre el temps que l'esdeveniment ha d'estar actiu per poder comprar, vendre o bescanviar entrades.

pendingDuration: Variable prefixada actualment en 10 minuts (600 segons), durant el qual el propietari d'una entrada pot bescanviar-la per l'accés a l'esdeveniment.

VALIDATOR_ROLE: Rol definit perquè certes adreces de confiança puguin fer el bescanvi de l'entrada per l'accés, únicament si l'entrada està en pending.

tickets: Mapping (clau – valor) que associa l'identificador d'una entrada amb un sturct de Tiquet determinat.

Mesures Implementades

Quan es van elaborar els objectius primaris, un dels principals era establir mesures preventives contra monopoli d'entrades, revenda massiva, inflació de preus, venda d'entrades ja bescanviades i compravenda d'entrades quan l'esdeveniment ha terminat, i a continuació es justificarà quines han sigut les mesures implementades dintre del Smart Contract per aconseguir cadascun dels subobjectius:

Prevenció contra el monopoli d'entrades

Com s'ha vist en la secció anterior, s'ha implementat la variables maxTicketsPerAddress, la qual assegura que un usuari no pugui comprar més entrades de les permeses. Aquesta restricció s'ha verificat en dos punts crítics com el procés de compra de tiquets en estoc (mintTickets) i en la compra d'entrades de revenda (buyTicket) de forma que cap adreça pot comprar més tiquets dels permesos.

Prevenció contra la revenda massiva

Per evitar que un usuari pugui revendre entrades de forma massiva, col·laboren dos variables: la primera, maxTicketsPerAddress que s'ha esmentat a l'apartat anterior i la segona ticketsPurchased i amb aquesta, verificant-se dintre de les funcions mintTickets i buyTicket, s'elimina la possibilitat que un usuari pugui crear un flux on compren el límit d'entrades per després revendre i tornar a començar el cicle. És per això que la variable ticketsPurchased guarda un històric del nombre d'entrades que ha tingut cada adreça durant la vida de l'esdeveniment, de forma que es corregeix aquest possible flux.

Prevenció sobre la inflació de preus

Per assolir aquest subobjectiu, s'ha hagut de qüestionar quin hauria de ser el límit de preu que pot posar un usuari a la seva entrada. La primera opció que vaig contemplar va ser que els usuaris no poguessin vendre l'entrada per un preu superior al valor original, però aquesta opció pot fer que els usuaris busquin una altra opció per vendre-la ja sigui perquè han tingut gestos externs relacionats a l'entrada que no poden aprofitar com la reserva d'una estança, o la congelació d'uns bitllets d'avió, etc. És per això que per a poder suplir aquesta possibilitat i que els usuaris no tractin buscar altres alternatives de venda, s'ha pres la decisió que els usuaris puguin vendre els seus tiquets com a màxim un 30% superior al preu de venda original i per a controlar-ho es verifica que el preu posar per l'usuari no superi el límit, de forma que en cas de fer-ho es reverteix la transacció avisant d'aquesta condició.

Protecció contra la venda d'entrades bescanviades o falses

Una de les mesures aplicades és que es comprovi en tot moment que una entrada estigui en l'estat de "Active" abans de poder fer qualsevol acció com un intent de venda. Aquesta restricció es troba implementada dintre de la funció addTicketsForSale la qual comprova abans de tot que l'estat no sigui ni "Pending" ni "redeemed", ja que per protegir als usuaris de moviments malintencionats, aquests estats limiten el que es pot fer amb l'entrada.

Protecció contra el comerç d'entrades fora de temps

En cas que el contracte tingui habilitada l'opció d'un límit de temps, s'ha establert que certes funcions no estiguin disponibles quan l'esdeveniment finalitza. Aquestes restriccions de temps s'han aplicat a funcions com la compra d'entrades d'estoc, la revenda d'entrades (únicament permet cancel·lar la venda per poder recuperar-la i així

transferir-la amb la funció transfer() de l'estàndard ERC721) o els canvis d'estat a "pending" i "redeemed" (el bescanvi de l'entrada per l'accés). Imposant aquestes restriccions en finalitzar l'esdeveniment, s'evita que els usuaris puguin pensar que estan comprant entrades per a un esdeveniment futur, i en realitat aquest ja ha passat. Per implementar-ho. S'han utilitzat dues variables, en primer lloc, el booleà useTimeLimit i , en segon lloc, la variable numèrica eventEndTime que ens diu quantes hores ha d'estar actiu l'esdeveniment.

Dependències Implementades

Tal com es va definir als objectius, un requisit per al Smart Contract és aplicar una sèrie d'estàndards i mesures preventives per a evitar possibles atacs. Actualment, existeixen una gran varietat d'atacs, sobretot, quan un Smart Contract manipula diners. És per això que necessitem que certes funcions que estiguin auditades per assegurar que algunes accions es realitzin de forma segura:

ERC721A.sol

Aquest Smart Contract agafa la implementació ja feta pel Smart Contracts de OpenZeppelin i aplica una sèrie de modificacions perquè sigui més eficient amb el Gas que utilitza a l'hora de generar les entrades, sobretot, quan el que vols és generar diverses entrades a la vegada. Si bé en el meu desenvolupament he fixat que sigui l'usuari qui pagui per aquesta generació, pot ser molt útil per persones que compren les entrades en grup perquè s'estalvia Gas i, per tant, els usuaris han de pagar menys. El seu funcionament radica en una sèrie d'assumpcions que fa com que l'ID sempre creixerà començant pel 0 i que en comptes de declarar l'estat de de un en un cada NFT generat, declara un únic estat per tot el grup de NFTs que hagi volgut generar l'usuari, de forma que estalvia Gas sobretot en transaccions on es comprin diverses entrades a la vegada.

Ownable.sol

Aquest contracte és utilitzat per aplicar el patró "ownable", és útil per aquelles funcions dintre del Smart Contract que només el propietari d'aquest pot llançar assegurant-nos que només hi haurà un únic owner. Alguns casos pràctics en els quals s'han aplicat són:

- addValidator()/removeValidator() → Ja que no volem que qualsevol persona pugui afegir/treure un validador, aquests han de ser persones designades per l'organitzador de forma que l'accés a l'esdeveniment sigui més controlat.
- setBaseURI() → No volem que es pugui posar una base del tokeURI diferent, pel fet que si no un atacant podria posar una adreça diferent de forma que l'usuari rebés informació errònia o que el tokeURI enviís a l'usuari a pàgines malicioses.
- withraw() → S'utilitza per a la funció de retirar diners, de forma que només el propietari del smart contract pugui retirar els fons aconseguits durant la venda d'entrades.

AccessControl.sol

Ens ajuda a poder establir rols diferents dintre del Smart Contract, en aquest cas, el rol del validador. D'aquesta forma no hem de donar més permisos dels necessaris, ja que si tots els validadors tinguessin el rol de Owner, podrien fer diverses accions

malicioses com retirar els diners recaptats de la venda, establir tokenURIs maliciosos, etc.

ReentrancyGuard.sol

Implementa una solució mes sofisticada per a evitar els atacs de reentrada, a més els Smart Contracts de Openzeppelin estan auditats per persones externes en termes de seguretat, és per això que he decidit delegar aquesta protecció a unes funcions més professionals.

Obervació: Els imports anteriors, han sigut canviats per les seves versions "Upgradeable", mes informació a l'apartat "Canvis en els objectius".

Strings.sol

S'utilitza per a la definició del tokenURI, com aquest en l'adreça fa servir l'identificador de l'entrada per guiar a l'usuari a la informació exacta sobre aquesta, necessitem convertir l'identificador numèric en una cadena de text. Per evitar errors de lògica i un correcte rendiment per estalviar còmput, és millor delegar aquesta funció a la funció que implementa aquest Smart Contract de Openzeppelin.

Clones.sol

S'utilitza dintre del contracte que fa de fàbrica, de forma que és genera un clon amb la implementació mínima necessària per fer les delegateCall al contracte clonat. Amb aquesta implantació s'ha buscar una reducció del bytecode de la fàbrica així com la reducció de gas quan es vol generar un Smart contract per a un nou esdeveniment.

Initializable.sol

S'utilitza dintre del contracte que conté la lògica de gestió d'esdeveniments. Implementa un sistema de control perquè una vegada el clon s'hagi inicialitzat amb el seu mètode "initialize", aquest no pugui tornar a inicialitzar-se, de la mateixa forma que un constructor no es pot cridar dues vegades. Per a fer això, OpenZeppelin proposa un control de versions, on es podrien definir diferents inicialitzadors, i a cadascun se li assigna una versió, una vegada aquella versió ha estat utilitzada (s'ha cridat a la funció que conté el initializer), ja no es pot tornar a fer servir. Aquesta metodologia seria més aprofitable si en algun moment s'ha d'inicialitzar altra vegada alguna variable, però donat el context del projecte, segurament no serà necessari encara que compleix perfectament el propòsit de protegir la funció pel qual s'utilitza.

Seguretat del Smart Contract

Gràcies als coneixements que he adquirit durant el desenvolupament de Smart Contracts amb la web "Solidty by example" he pogut contrastar quines de les 18 principals amenaces recollides a la web anterior podrien afectar a aquest treball, l'objectiu d'aquest apartat és representar quins vectors d'atac poden afectar el desenvolupament, determinar si és un risc assumible o s'ha de corregir i, en cas de corregir-ho, quines són les contramesures aplicades:

Atac de reentrada:

Un dels atacs més repetitius i comuns en els Smart Contrcats són els atacs de "reentrancy". Quan una persona invoca una funció dintre d'un Smart Contract, aquesta

funció pot no actualitzar de forma immediata les variables globals fins que no s'acaba completament la seva execució. Això permet als atacants aprofitar aquest retard per repetir l'acció i obtenir un benefici.

Per exemple, en el Smart Contrcat desenvolupat, suposem la funció "mintTickets()". Aquesta funció genera tants tiquets com demana l'usuari, i en cas que pagui més que el valor establert, el contracte retorna automàticament el sobrant. Un atacant que vulgui explotar aquesta funció podria crear un Smart Contract maliciós que invoqui a mintTickets() i, tan aviat com el Smart Contract de l'atacant detecta l'ingrés del sobrant, torni a invocar a la funció mintTickets(). Si en fer aquesta segona invocació, les variables globals encara no estan actualitzades (per exemple, el nombre de tiquets que ha comprat l'adreça), pot aprofitar per tornar a comprar i obtenir el doble d'entrades del permès, ja que el Smart Contract no ha pogut comprovar amb les variables actualitzades de la primera execució si l'usuari ja havia comprat fins al límit de tiquets per adreça.

De la mateixa manera, es pot aplicar aquest atac a la resta de funcions que retornen diners. Per a evitar-ho, s'utilitzen els "reentrancy guards". Els quals són una funció petita que actualitza l'estat de la funció com "en execució" amb un booleà a "true". Continuant amb l'exemple de mintTickets(), el reentrancy guard s'executaria just abans d'executar la funció, de forma que marca la funció com "en execució" posant el booleà a "true" i el retrona a "false" quan es completa tota l'execució, assegurat així que les variables globals també estan actualitzades. D'aquesta manera si un atacant torna a invocar a la funció durant l'execució abans que es guardin els canvis de la primera execució, el primer a executar-se serà el reentrancy guard, aquest detectarà que la funció ja està en execució gràcies al booleà que s'havia posat a "true" i revertirà aquesta segona crida. D'aquesta forma és com es protegeixen les funcions crítiques com mintTickets, buyTicket, o withdraw contra possibles atacs de reentrncy.

Front Running

Primerament per entendre aquest tipus d'atac s'han de definir dos conceptes, ja que quan un usuari vol enviar una transacció a la xarxa, ha d'especificar dues coses les quals estan relacionades amb el gas:

En primer lloc, el priority fee, que és la propina que s'emporta el validador per incloure la transacció en un bloc. Aquesta va lligada a la quantitat de gas que es necessiti per fer la transacció, és a dir, si la nostra priority fee és de 2Gwei i la transacció consumeix 5 unitats de Gas, el validador s'emportarà 10Gwei.

En segon lloc, tenim el "max fee", que és el cost total que l'usuari està disposat a pagar per unitat de Gas, aquesta quantitat ha d'englobar la tarifa base (que és el preu per unitat de gas establert en el bloc) i la priority fee. Cal aclarir que aquesta base fee no se l'emportà ningú sinó que es "crema". Per posar un exemple sobre aquest concepte, si el priority fee establert és de 2Gwei per unitat de gas consumida, i el max fee és de 10Gwei per unitat de gas consumida, llavors quedarien 8Gwei per tractar de cobrir aquest base fee. Si per exemple aquest max fee és menor que la suma del base fee + priority fee, llavors la transacció no es realitza, i l'usuari pot, retallar import sobre la priority fee, de forma que al pagar menys al validador, pot no incloure-la o incloure-la en un bloc més tard, o bé augmentar el max fee per a poder cobrir el base fee.

L'atac de front running es produeix quan un atacant vol que la seva transacció es faci primer que la d'un altre que l'havia fet amb anterioritat, per a realitzar-ho, l'atacant pot posar una priority fee més elevada a la seva transacció, de forma que aquesta

s'inclogui abans al bloc. Amb aquest atac, un usuari podria tenir una "prioritat" a l'hora de comprar una entrada per a un esdeveniment, de forma que si és probable que s'esgotin les entrades, s'asseguri que la seva transacció de compra arribarà abans que les fetes per altres usuaris.

Aquest risc és inherent per la mateixa xarxa d'Ethereum i en aquest cas s'hauria d'acceptar.

Block Timestamp Manipulation:

Aquest tipus consisteix que un validador avanci o endarrereixi lleugerament (ja que tenen un límit) el timestamp del bloc, d'aquesta forma si el Smart Contract d'un esdeveniment té establert un temps de finalització, podria avançar uns segons aquest final de l'esdeveniment.

Tenint en compte aquest atac, si bé és veritat que pot existir un cert risc de manipulació, els validadors només el poden modificar el temps d'un bloc amb una folgança d'uns segons, és per això que considero que el risc és baix i s'ha d'acceptar.

Objectius completats

Finalment, durant el desenvolupament d'aquesta primera fase, dintre del Smart Contrcat s'han assolit tots els Objectius primaris que es van esmentar al informe inicial:

- Permetre la compra d'entrades NFT en estoc.
- Permetre la venda d'entrades NFT.
- Permetre la consulta d'entrades NFT a la venda al mercat amb el seu preu.
- Permetre la compra d'entrades NFT que estiguin venent altres usuaris.
- Establir mesures preventives contra monopoli d'entrades, revenda massiva, inflació de preus, venda d'entrades ja bescanviades i compravenda d'entrades quan l'esdeveniment ha terminat.
- Complir amb els estàndards de seguretat per a evitar possibles vectors d'atac.
- Complir amb els estàndards dels NFT com el ERC721[1] i el ERC165[2]
- Poder consultar la legitimitat d'una entrada.

També s'han aconseguit dos dels cinc objectius secundaris que s'havien planificat:

- Implementació avançada del codi perquè les transaccions siguin eficients pel que fa a la despesa de Gas: Això ha sigut gràcies a la implementació del estàndard ERC721A.
- Establiment de rols com verificadors (un verificador seria l'operari encarregat de verificar la pertinença de l'entrada i bescanviar-la per l'accés a l'esdeveniment): Aquest objectiu s'ha complert mitjançant l'ús de AccessControl.sol.

[1] **ERC-721**: Conjunt de regles establertes per a que es puguin manipular els NFT entre aplicacions i Smart Contracts.

[2] **ERC-165**: Regla que defineix un mètode per a que els Smart Contrats detectin i publiquin quins altres estàndards compleixen.

Canvis en els objectius

Com ja s'ha parlat dintre de la metodologia aplicada, un dels requisits determinants per a la selecció d'una eina és que comptés amb la possibilitat d'implementar tests automàtics per a poder avaluar el correcte funcionament del Smart Contract. I això és perquè l'execució de proves automatitzades s'ha redefinit com un objectiu primari vital per a poder comprovar el correcte comportament del Smart Contract.

Amb aquesta finalitat, s'ha revisat la documentació de Hardhat juntament amb els exemples que hi ha i s'han elaborat un total de 38 tests, dissenyats per a validar tant el correcte funcionament dels mètodes implementats en escenaris d'èxit, com la gestió adequada de situacions en què es produeixin intents de transaccions no permeses o que impliquin valors límit i frontera.

D'altra banda, s'ha considerat que la Dapp hauria de ser capaç d'interactuar amb diferents Smarts Contracts, cosa que permet la gestió simultània de les entrades de diversos esdeveniments.

Com aquest objectiu no s'havia plantejat, la solució inicial que s'ha proposat és l'ús d'un patró de desenvolupament que es va aprendre a l'assignatura de "Disseny del Software", concretament, el patró de fàbrica. Aplicant-ho a aquest treball, dona com a resultat un nou Smart Contract que fa de fàbrica, desplegant el Smart Contract de l'esdeveniment. Però això va presentar un nou problema que ha provocat un canvi en el paradigma d'aquest projecte: la mida límit del bytecode.

Per posar en context aquest problema, Ethereum ha tingut diferents Hard forks[3] al llarg de la seva vida. En aquest cas, el projecte s'ha vist limitat per un hard fork que es va implementar en novembre de 2016, conegut com a "*Spurious Dragon hard-fork*". On, entre altres implementacions, es va introduir el EIP-170[4], que imposa un límit en la mida del contracte com a contra mesura als atacs de denegació de serveis (Dos) que patia la xarxa. Aquest límit, va ser establert en 24576 bytes, el qual, al fer el desenvolupament de la fàbrica que importa i guarda una instància completa del contracte d'esdeveniments per cadascun dels esdeveniments, va sobrepassar aquest límit, tal i com es pot apreciar a la següent captura:

```
Warning: Contract code size is 26921 bytes and exceeds 24576 bytes (a limit introduced in Spurious Dragon).
 This contract may not be deployable on Mainnet. Consider enabling the optimizer (with a low "runs" value!)
, turning off revert strings, or using libraries.
 --> contracts/EventTicketFactory.sol:6:1:
```

Com es pot veure, a l'haver implantat el patró de fàbrica, se supera el límit per 2345 bytes.

Fent una primera recerca de com es podria resoldre, vaig trobar com a solució inicial utilitzar optimitzacions en la compilació, ja que dintre del fitxer de configuració de Hardhat es pot habilitar indicant també el número de vegades que es preveu que s'executarà el codi. Això permet al compilador entendre que, si es posa un número elevat d'execucions, s'ha de centrar a optimitzar el codi per a reduir el consum de gas

[3] **Hard Fork**: Canvi en les regles de consens que no és compatible cap enrere. Provoquen una bifurcació que pot dividir la xarxa en els usuaris que segueixen les antigues regles, o aquells que es regeixen per les noves. Aquests són problemàtics, ja que el còmput que estava unificat, se separa i una de les xarxes pot acabar perdent el seu valor.

[4] **EIP**: Són les sigles per (Ethereum Improvement Proposal), documents formals que proposen canvis per millorar Ethereum que, si s'accepten per la comunitat, deriven en estàndards.

de les transaccions, de forma que, utilitzarà més operacions (mida del contracte més gran), però aquestes instruccions poden tenir menys cost computacional, reduint el consum de Gas per part de les transaccions de l'usuari.

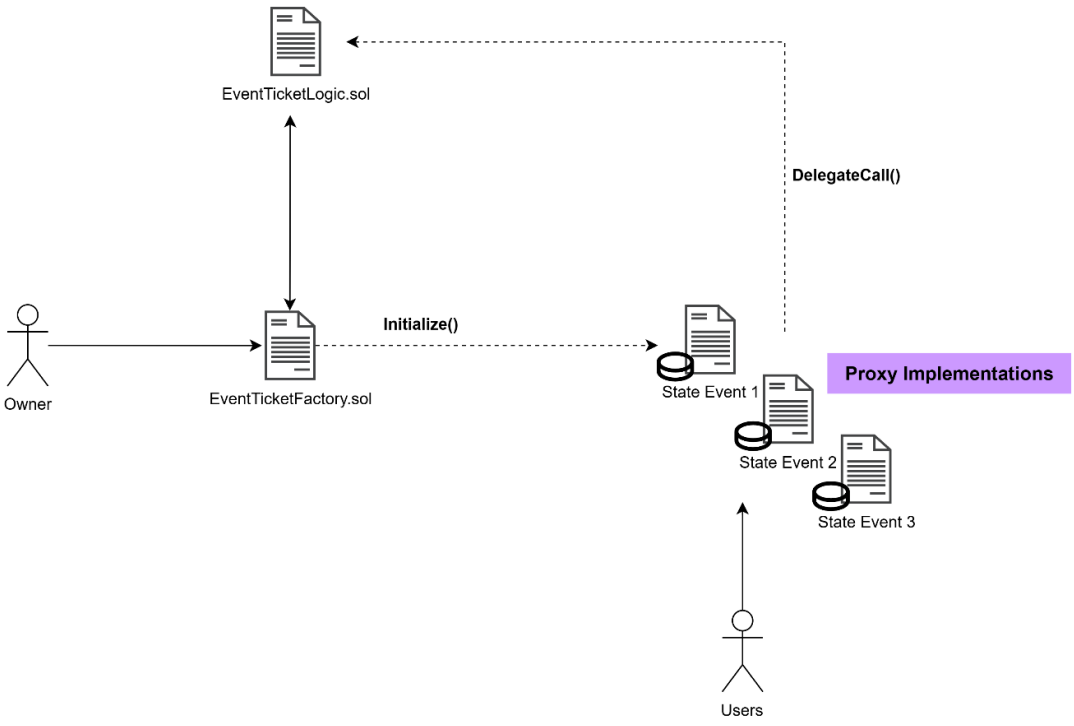
En canvi, si s'indica al compilador que hi haurà número d'execucions baixes, se centra a reduir la mida del Smart Contract per a fer desplegaments més eficients (de menor espai) utilitzant menys instruccions, però computacionalment més costoses, fet que incrementaria el Gas de cadascuna de les transaccions dels usuaris.

Si bé aquesta és una solució còmoda, un dels objectius proposats és l'eficiència de les transaccions i és per això que s'ha fet una recerca sobre altres solucions.

Finalment, i a partir de la revisió de l'article "Downsizing contracts to fight the contract size limit" publicat a ethreum.org, s'ha identificat una solució potencial: els "Proxies". En concret el EIP-1167 que introdueix el concepte de Minimal Proxy.

La idea sota aquest patró és la creació de petits Smart Contracts que, en comptes d'incloure tota la lògica de funcionament, deleguen l'execució a un Smart Contract que sí que conté tota la lògica, mentre que els clons (proxy) només emmagatzemen una referència a aquest contracte. Això s'aconsegueix mitjançant l'ús de la implementació de "Clones" de la llibreria de OpenZeppelin on els clones generats utilitzen la instrucció delegateCall, la qual fa que tot el processament es faci dintre del Smart Contract que conté la lògica, però fent servir les variables globals del contracte clonat. Com ha resultat, cada clon només ha de guardar la informació específica del seu esdeveniment i l'adreça al contracte que té la lògica.

Per entendre millor el concepte, es presenta la següent figura:



Com es pot observar, els usuaris interactuarien amb el proxy, el qual crida mitjançant la funció deleagateCall al contracte que conté la lògica. De forma que se separa la gestió de la lògica que és comuna per tots els esdeveniments, de l'emmagatzematge i les regles imposades per cada esdeveniment.

Per a implementar aquest patró s'ha hagut de canviar les llibreries ERC721A, Ownable, AccessControl i ReentrancyGuard a la seva versió "Upgradeable". Aquest canvi ha sigut necessari, ja que les versions "Upgradeable" permeten la inicialització de l'estat a través d'una funció diferent que no sigui el constructor. Això és fonamental per aquest patró perquè els clons com són una "copia" d'un contracte, no activen el constructor. A més, tampoc podem utilitzar el constructor al contracte que té la lògica, ja que no volem inicialitzar un estat. Si inicialitzéssim un estat determinat per la lògica, tots els proxies tindrien aquest estat, i per tant tots els proxies tindrien el mateix preu per entrada, el mateix nombre d'entrades màximes, compartirien identificadors entre esdeveniments, etc.

Per aquesta raó, com que cada esdeveniment ha de tenir les seves pròpies variables, s'utilitza una funció inicialitzadora que fa de "pseudo constructor" la qual podem cridar quan es vulgui. Això dona més flexibilitat, però amb la conseqüència de què s'ha d'assegurar que només es cridi una única vegada (ja que no volem inicialitzar dos veges un esdeveniment), i aquí és on juga un paper essencial la llibreria "Initializable" de OpenZeppelin, perquè s'encarrega de protegir l'inicialitzador, garantint que no es pot cridar cap vegada més.

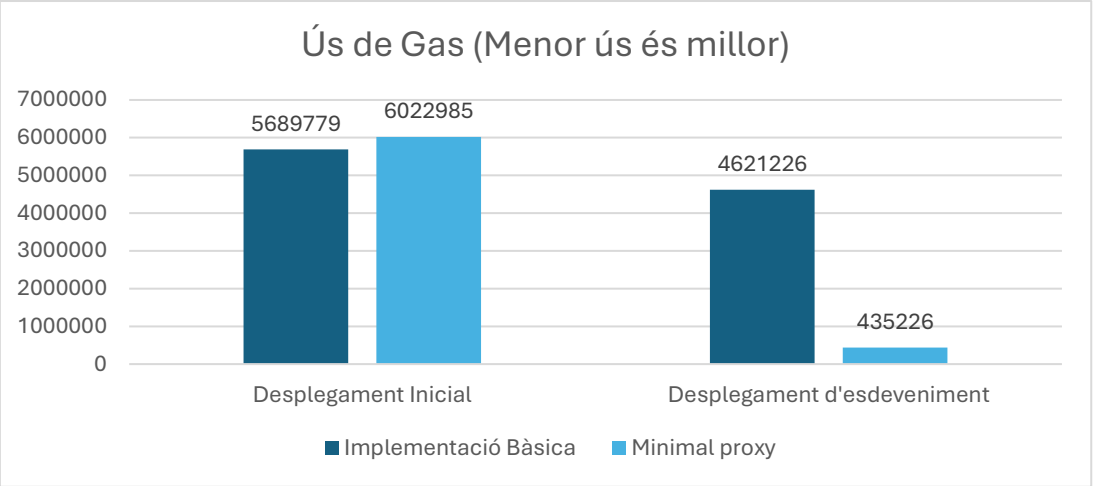
Observació: Com el contracte que conté la lògica no té cap estat definit, el constructor s'utilitza per bloquejar la funció "initialize" de forma que un actor maliciós no pugui donar-li un estat, el qual podria provocar bugs o errors imprevisibles.

Finalment, la implementació d'aquest patró ha donat com a resultat un bytecode de 24269 bytes enfront dels 26921 bytes anteriors, és a dir, una reducció d'un 9,85% (2652 bytes), deixant el Smart Contract dintre de la mida límit establerta.

A més, ha produït una altra gran millora, ja que l'anterior versió donava un cost inicial de desplegament d'uns 5689779 unitats de Gas, on per cada esdeveniment que es volia llençar tenia un cost de 4621226 unitats de gas.

Mentre que aquesta nova versió gasta 6022985 unitats de gas pel desplegament inicial, mentre que per cada esdeveniment (proxy) a llençar es gasta uns 435226 unitats de gas.

Això deixa com a resultat que aquesta nova versió ha augmentat un 5,86% el cost del primer desplegament (la fabrica amb el contracte que conté la lògica) però s'ha aconseguit que el desplegament de cada esdeveniment utilitzi un 90,58% menys de gas que l'anterior versió, una eficiència molt gran tenint en compte que la fàbrica podria tenir milers d'esdeveniments, la qual es pot observar gràficament al següent gràfic:



Gràcies a aquesta nova funcionalitat de minimal Proxy combinat amb el patró de fàbrica, no s'haurà de reconfigurar una Dapp per cada esdeveniment, sinó que es tindrà tot integrat en un mateix lloc, cosa que millora l'experiència d'usuari.

En conclusió, d'aquesta part, i segons els canvis realitzats, els objectius restants són:

Objectius primaris

- Desenvolupar una aplicació descentralitzada que pugui interactuar amb el Smart Contract de forma que faci de pont/interfície entre el Wallet del usuari i el Smart Contract.

Objectius secundaris

- Creació de codis QR per a una gestió d'accés als esdeveniments més fluida.
- Creació del token URI i implementació a un IPFS per afegir més informació a l'entrada (nom de l'esdeveniment, data i hora, preu de compra, informació postvenda...).
- Llençar el Smart Contract a una Testnet d'Ethereum sortint així de l'entorn local.
- Donar suport a través de la Dapp per poder gestionar diferents esdeveniments a l'hora.

Planificació del treball: Seguiment I

Anàlisi de la Primera fase

Continuant amb la planificació de l'informe inicial, les tasques que van ser planificades per a fer durant aquest informe son:

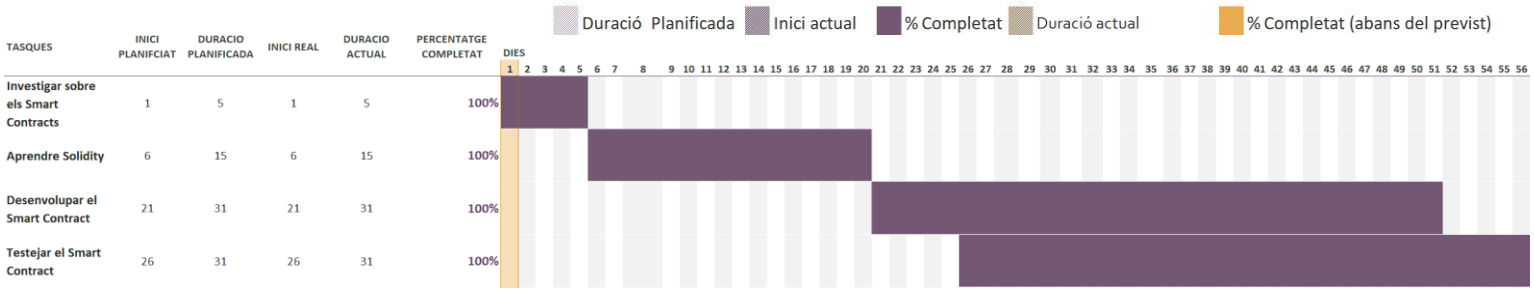
- Investigar sobre els Smart Contracts (15 de febrer – 19 de febrer)
- Aprendre Solidity (20 de febrer – 6 de març)
- Desenvolupar el Smart Contract (7 de març – 6 d'abril)
- Testejar el Smart Contract (14 de març – 13 d'abril)

Tant la part d'investigació sobre els Smart Contracts com la tasca d'aprendre Solidity s'han complert dins de les dates establertes sense cap inconvenient.

D'altra banda, durant la tasca del desenvolupament del Smart Contract s'han trobat alguns obstacles que han pogut posar en perill el compliment dintre de les fites. Els més rellevants han sigut les decisions que s'han pres durant el desenvolupament, reflectit en el següent apartat de consideracions, o el nou objectiu que ha comportat la implementació de la fàbrica i el patró Minimal Proxy.

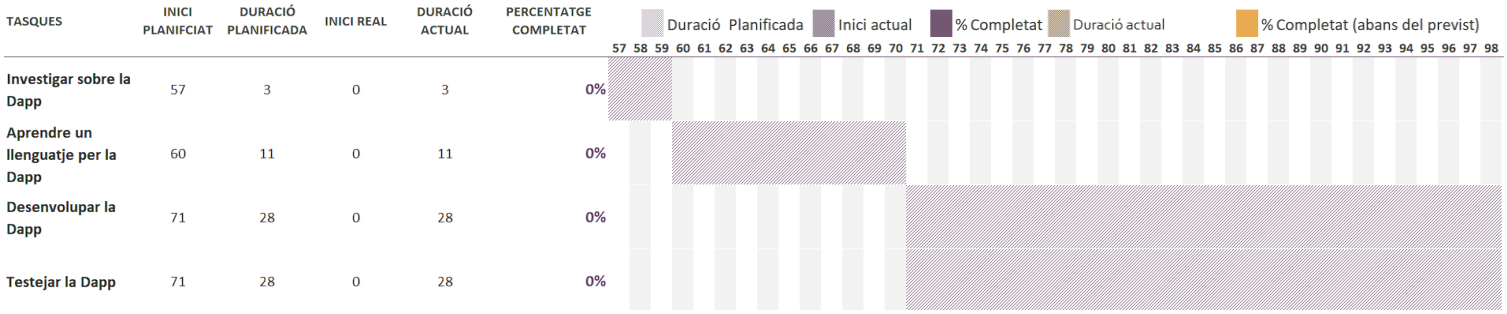
Per últim, el testeig ha tingut un transcurs consistent acompanyant al desenvolupament de les funcions introduïdes al Smart Contract.

Gràficament aquesta part : [Dia 1 = 15 de febrer i Dia 56 = 13 d'abril]



Segona Fase: Preparació

Aquesta Segona Fase que es veurà reflectida dintre del següent informe de progrés, compren des del 14 d'abril de 2025 fins al 25 de maig de 2025. [Dies 57 a 98]



Aquesta segona part estarà més enfocada en un procés creatiu amb tasques centrades en el desenvolupament de l'aplicació distribuïda que interactuarà amb el Smart Contract desenvolupat. Les dates i tasques planificades per aquesta part són les següents:

- Investigar sobre la Dapp (14 d'abril – 16 d'abril): Aquesta part se centrarà a buscar projectes i referències per a posteriors decisions com el framework a utilitzar, quines bones pràctiques s'han de considerar, etc.
- Aprendre un llenguatje per la Dapp (17 d'abril -27 d'abril): Durant aquesta part se centrarà a aprendre la tecnologia amb la qual es farà la part central d'aquesta segona part.
- Desenvolupar la Dapp (28 d'abril – 25 de maig): La fase consistirà en el mateix desenvolupament de l'aplicació distribuïda, aportant idees, esquemes de colors a utilitzar i una visió més "visual" pel projecte.
- Testejar la Dapp (28 d'abril – 25 de maig): Aquesta part se centrarà a comprovar aspectes com la usabilitat, el rendiment, la funcionalitat, de l'aplicació desenvolupada, de forma que en següents fases es pugui fer un refinament.

Consideracions

Concepte de transferència d'una entrada

La idea inicial és que les transaccions d'entrades es puguin fer a través del Smart Contarct, però, en utilitzar el ERC721, sempre es defineix una funció transfer() la qual permet transferir un NFT d'una adreça a una altra. Llavors em vaig plantejar, hauria de limitar aquesta funció perquè els usuaris únicament puguin vendre les seves entrades a través de la meva implementació? O hauria de permetre que ells mateixos puguin transferir a plaer les seves entrades?

Inicialment, vaig pensar que el fet de deixar la funció transfer() sense limitar, faria que qualsevol persona pugui vendre la seva entrada de forma externa, sobrepassant la contramesura proposada per evitar la inflació de preus. Si bé, és veritat, també s'ha de considerar que la transferència contemplada per mi és aquella en la qual es busca que s'aprofiti l'entrada, però pot haver-hi un cert col·leccionisme d'entrades ja bescanviades, ja sigui com a commemoració d'un determinat esdeveniment, o una persona que vol moure les seves entrades a una altra adreça. A més, la filosofia darrera dels NFT es caracteritza per la llibertat de fer el que vulguis amb un Token que és teu, ja que, al cap i a la fi, has pagat per ell. És per això, que per poder continuar amb aquesta filosofia, però tractant de disminuir el seu ús, he considerat deixar que les persones puguin aprofitar totes les característiques que implementa el ERC721 inclosa les transferències. Però, i com a contra mesura per no incentivar-la, aquesta interacció no estarà implementada dintre de la distributed application que faré. De forma que, els usuaris tècnics o amb suficients coneixements tinguin aquesta possibilitat, mentre que l'usuari Estàndard, que per norma, no es preocuparà d'això, ja que per part del comprador, aquest només vol assegurar-se de poder comprar una entrada a un preu raonable sense haver de confiar en el venedor, mentre que el venedor únicament vol recuperar la seva inversió, perquè no pot aprofitar l'entrada (sigui perquè no pot anar o sigui per qualsevol assumpte extern). D'aquesta forma, se satisfan ambdues parts.

Concepte de validació d'una entrada

El validador ha de declarar l'entrada com bescanviada, això vol dir que ha de pagar un cert gas, per solucionar això hi hauria dues opcions:

- Podria fer que el validador enviés per exemple un QR a l'usuari perquè aquest pugui posar el seu tiquet com a bescanviat (i el validador ho comprova per deixar passar a l'usuari) però això vol dir que hi ha d'haver un doble intercanvi d'informació el que suposa el doble de temps en donar accés a una persona, i la prioritat seria que els usuaris puguin accedir a l'esdeveniment de forma ràpida.

- Fer un servidor centralitzat que s'encarregui de la interacció amb el client per autoritzar el client a bescanviar la seva entrada i l'usuari finalment entra a l'esdeveniment. Però aquest mètode suposa desenvolupar un element que centralitza les interaccions, contradient la filosofia distribuïda d'aquest projecte.

Havent analitzat les dues opcions anteriors, crec que és millor una tercera opció on es tractaria de posar un fons suficient en cada adreça validadora (ja que normalment

seran màquines automàtiques) i estimar el preu per incloure-ho a l'entrada en concepte de gestió. Tanmateix, i tenint en compte el preu de la Mainnet d'Etehreum, l'opció òptima (econòmicament parlant) seria desplegar el contracte en una side-chain que permeti l'intercanvi amb Ethers o que els Usuaris puguin fer un pagament comú en targeta per pagar l'entrada bescanviant per sota els diners per la moneda de la cadena seleccionada.

S'ha de tenir en compte que això és un pensament més empresarial del projecte que es podria desenvolupar. Però para fins acadèmics i de recerca previstos per aquest treball, s'utilitzarà una xarxa local d'Ethereum o, en última instància, una testnet.

Concepte d'estat dels tiquets

Els Smart Contracts no tenen una forma per establir esdeveniments que es realitzin de forma automàtica sense que ningú faci res. És per això que, quan es posa un tiquet en l'estat de "Pending", quan passa el temps corresponent no hi cap forma de fer que el Smart Contrcat de forma automàtica canviï l'estat del tiquet a "Active". És per això que, quan un usuari que hagi posat el seu tiquet en "Pending" i vulgui realitzar alguna acció que no sigui bescanviar el tiquet per l'entrada a l'esdeveniment, he dissenyat una funció que es posa dintre de cada una d'aquestes funcions "no permeses" quan el tiquet està en "Pending" per poder comprovar si el temps de "Pending" ha vençut, de forma que aquesta funció canviï l'estat del tiquet a "Active" per poder continuar amb la petició del client. Aquesta funció s'ha afegit a la lògica de funcions com addTicketsForSale() i buyReselledTickets().

Concepte de venda

Quan un usuari decideix vendre la seva entrada seria perillós que fos ell mateix qui té l'entrada, ja que s'hauria de confiar en la voluntat de l'usuari en transferir l'entrada una vegada s'ha rebut el pagament. A més, aquest fet no encaixa amb la filosofia de no haver de confiar en ningú i és per això que s'ha implementat un sistema on el propi Smart Contract sigui qui rebi el tiquet per vendre de forma automàtica quan un usuari efectua el pagament de l'entrada. Si l'usuari vol cancel·lar la venda té a la seva disposició la funció cancelTicketForSale() on el Smart Contract revisa que el remitent de la petició sigui la mateixa adreça que es va guardar a l'atribut "seller" del tiquet.

Aclariment sobre el ERC721

Quan volem transferir un token NFT que utilitza el ERC721, si fem servir l'operació, safeTransferFrom(), aquesta funció, en cas que el recipient sigui un Smart Contract, truca a la funció onERC721Received() que ha de tenir implementada el Smart Contract. Aquesta trucada la fa per poder saber si el Smart Contract sap com manipular aquests tipus de tokens. De forma que si no l'implementa, es reverteix la transferència. Això dona una capa de seguretat a la transferència perquè ens indica que el smart Contract té la capacitat de manipular correctament el NFT i aquest no es quedarà bloquejat. És per això que he implementat la funció onERC721Received() dintre del Smart Contract perquè per poder fer les vendes aquest ha de tenir la capacitat de ser el "propietari" de les entrades, per poder fer la transferència quan una persona paga per una entrada que està en revenda.

Bibliografia

HARDHAT

Hardhat Network | Ethereum development environment for professionals by Nomic Foundation [en línea], (sin fecha). Hardhat | Ethereum development environment for professionals by Nomic Foundation. [Consultat el 17 de febrer de 2025]. Disponible en: https://hardhat.org/hardhat-network/docs/overview

Testing contracts | Ethereum development environment for professionals by Nomic Foundation [en línea], (sin fecha). Hardhat | Ethereum development environment for professionals by Nomic Foundation. [Consultat el 17 de febrer de 2025]. Disponible en: https://hardhat.org/hardhat-runner/docs/guides/test-contracts

FOUNDRY

GitHub - foundry-rs/foundry: Foundry is a blazing fast, portable and modular toolkit for Ethereum application development written in Rust. [en línea], (sin fecha). GitHub. [Consultat el 17 de febrer de 2025]. Disponible en: https://github.com/foundry-rs/foundry?tab=readme-ov-file

Foundry Book [en línea], (sin fecha). Introduction - Foundry Book. [Consultat el 17 de febrer de 2025]. Disponible en: https://book.getfoundry.sh/

REMIX IDE

Bienvenido a la documentación de Remix — documentación de Remix - Ethereum IDE - 1 [en línea], (sin fecha). Welcome to Remix's documentation! — Remix - Ethereum IDE 1 documentation. [Consultat el 17 de febrer de 2025]. Disponible en: https://remix-ide.readthedocs.io/es/latest/

IMPORTS DEL CODI

Aututu, V., (2024). Understanding ERC721A Standard [en línea]. Medium. [Consultat el 8 de marzo de 2025]. Disponible en: https://ututuv.medium.com/understanding-erc721a-standard-92f6b30ad16b

Azuki [en línea], (sin fecha). Azuki. [Consultat el 8 de març de 2025]. Disponible en: https://www.azuki.com/erc721a

ERC721A Documentation [en línea], (sin fecha). Site not found · GitHub Pages. [Consultat el 8 de març de 2025]. Disponible en: https://chiru-labs.github.io/ERC721A/#/

Access Control - OpenZeppelin Docs [en línea], (sin fecha). Documentation - OpenZeppelin Docs. [Consultat el 8 de març de 2025]. Disponible en: https://docs.openzeppelin.com/contracts/5.x/access-control

Utilities - OpenZeppelin Docs [en línea], (sin fecha). Documentation - OpenZeppelin Docs. [Consultat el 8 de març de 2025]. Disponible en: https://docs.openzeppelin.com/contracts/5.x/api/utils#security

Conquer Blocks, (2023). ATAQUE más común en SMART CONTRACTS (Reentrancy Attack) ✚ Cómo PROTEGER un Contrato Inteligente [en línea]. YouTube. [Consultat el 9 de març de 2025]. Disponible en: https://www.youtube.com/watch?v=MxxwxHc2pmM

Utilities - OpenZeppelin Docs [en línea], (sin fecha). Documentation - OpenZeppelin Docs. [Consultat el 31 de març de 2025]. Disponible en: https://docs.openzeppelin.com/contracts/5.x/utilities

Clones - OpenZeppelin Docs [en línea], (sin fecha). Documentation - OpenZeppelin Docs. [Consultat el 2 d'abril de 2025]. Disponible en: https://docs.openzeppelin.com/contracts/5.x/api/proxy#Clones

Proxies - OpenZeppelin Docs [en línea], (sin fecha). Documentation - OpenZeppelin Docs. [Consultat el 2 d'abril de 2025]. Disponible en: https://docs.openzeppelin.com/contracts/5.x/api/proxy#Initializable

STANDARS

Ethereum Improvement Proposals [en línea], (sin fecha). Ethereum Improvement Proposals. [Consultat el 8 de març de 2025]. Disponible en: https://eips.ethereum.org/EIPS/eip-721

APRENENTATGE: SOLIDITY I SMART CONTRACTS

Andreas M. Antonopoulos and Gavin Wood Ph.D (2018) Mastering Ethereum : Building Smart Contracts and DApps. Sebastopol, CA: O'Reilly Media. [Consultat el 20 de febrer de 2025]. Disponible en: https://research.ebsco.com/linkprocessor/plink?id=fcbc8282-5c2e-34a7-bde8-805e7ab76a41

Security Considerations — Solidity 0.8.29 documentation [en línea], (sin fecha). Solidity — Solidity 0.8.29 documentation. [Consultat el 20 de febrer de 2025]. Disponible en: https://docs.soliditylang.org/en/stable/security-considerations.html#use-the-checks-effects-interactions-pattern

Solidity by Example [en línea], (sin fecha). Solidity by Example. [Consultat el 21 de febrer de 2025]. Disponible en: https://solidity-by-example.org/

CANVIS EN ELS OBJECTIUS

5. Testing contracts | Ethereum development environment for professionals by Nomic Foundation [en línea], (sin fecha). Hardhat | Ethereum development environment for professionals by Nomic Foundation. [Consultat el 21 de febrer de 2025]. Disponible en: https://hardhat.org/tutorial/testing-contracts

Compiling your contracts | Ethereum development environment for professionals by Nomic Foundation [en línea], (sin fecha). Hardhat | Ethereum development environment for professionals by Nomic Foundation. [Consultat el 31 de març de 2025]. Disponible en: https://hardhat.org/hardhat-runner/docs/guides/compile-contracts#build-info-files

History and Forks of Ethereum [en línea], (sin fecha). ethereum.org. [Consultat el 31 de març de 2025]. Disponible en: https://ethereum.org/en/history/#spurious-dragon

EIP-170: Contract code size limit [en línea], (sin fecha). Ethereum Improvement Proposals. [Consultat el 31 de març de 2025]. Disponible en: https://eips.ethereum.org/EIPS/eip-170

Downsizing contracts to fight the contract size limit [en línea], (sin fecha). ethereum.org. [Consultat el 1 d'abril de 2025]. Disponible en: https://ethereum.org/en/developers/tutorials/downsizing-contracts-to-fight-the-contract-size-limit/

Ethereum Improvement Proposals [en línea], (sin fecha). Ethereum Improvement Proposals. [Consultat el 1 d'abril de 2025]. Disponible en: https://eips.ethereum.org/EIPS/eip-1167

Security, O., (2019). Deep dive into the Minimal Proxy contract - OpenZeppelin blog [en línea]. OpenZeppelin Blog. [Consultat el 1 d'abril de 2025]. Disponible en: https://blog.openzeppelin.com/deep-dive-into-the-minimal-proxy-contract

ERC721A Documentation [en línea], (sin fecha). Site not found · GitHub Pages. [Consultat el 2 d'abril de 2025]. Disponible en: https://chiru-labs.github.io/ERC721A/#/upgradeable?id=usage

Using with Upgrades - OpenZeppelin Docs [en línea], (sin fecha). Documentation - OpenZeppelin Docs. [Consultat el 2 d'abril de 2025]. Disponible en: https://docs.openzeppelin.com/contracts/5.x/upgradeable