Universitat de Girona
Escola Politècnica Superior

Projecte fi de grau

Estudi: Grau en Enginyeria Informàtica

Títol: Blockchain explorer de la Teranyina

Document: Memòria

Alumne: Marc Bramon Tarrés

Tutor: Jose Luis de La Rosa Esteva
Departament: ENGINYERIA ELÈCTRICA, ELECTRÒNICA I AUTOMÀTICA
Àrea: ENGINYERIA DE SISTEMES I AUTOMÀTICA

Convocatòria (mes/any): Juny 2022

Universitat de Girona
Escola Politècnica Superior

PROJECTE FI DE GRAU

Blockchain explorer de la Teranyina

Autor:
Marc BRAMON TARRÉS

Juny 2022

Grau en Enginyeria Informàtica

Tutors:
Jose Luis DE LA ROSA ESTEVA

Índex

CAPÍTOL 1

Introducció

Aquest projecte té com a finalitat construir un explorador de la blockchain Teranyina, mostrant les seves dades més rellevants d'una manera més visual i amb una barra de cerca que permetrà buscar per diferents elements de la blockchain.

La blockchain Teranyina és una cadena de blocs que permet gaudir d'una plataforma basada en blockchain sobre la qual els investigadors, universitats i centres de recerca poden desenvolupar els seus projectes i fer les proves necessàries amb un cost molt reduït o gratuït gràcies als faucet tokens: criptomonedes que els usuaris obtindran de manera gratuïta, i que els permetran fer transaccions en la xarxa de Teranyina, que tindrien un cost en qualsevol altra blockchain.

Un explorador blockchain és com un motor de cerca que revela informació sobre l'estat passat i actual d'una blockchain. Això pot ser útil quan es vol fer un seguiment del progrés d'un pagament específic o comprovar el saldo i l'historial d'una adreça. Qualsevol persona amb una connexió a Internet pot utilitzar un explorador per veure totes les transaccions d'una blockchain pública.

Per què necessitem un explorador si les blockchains són, en el fons, bases de dades distribuïdes?

- Amb un blockchain explorer, el procés d'afegir blocs a la nostra blockchain es fa visible.
- Per obtenir dades d'una blockchain és necessari una API que sàpiga interpretar la informació que ens arriba criptografiada de la blockchain. No es comporta com una base de dades convencional.
- Ens assegura rapidesa a l'hora de buscar informació detallada sobre la nostra blockchain.

  A vegades, per exemple, pot arribar a ser complicat arribar a obtenir tota la informació que té un bloc. Mitjançant les APIs que ens ajuden a extreure aquesta informació és possible que necessitem diverses consultes per reunir tota aquesta informació.

1.1 Motivació

El món de les blockchains és un món relativament nou i interessant. Durant el transcurs de la carrera no se m'havia presentat l'oportunitat de poder-me

endinsar en aquest món, que des de fora em semblava tan interessant.

Quan vaig veure les diverses propostes que el centre EASY oferia a través de la pàgina de propostes de Projectes de fi de grau de la UdG, no ho vaig dubtar.

El projecte de fer un explorador per la seva blockchain Teranyina és el que em va cridar més l'atenció, ja que barrejava conceptes nous per mi, com són els relacionats amb el món de les blockchains i em donava la possibilitat de crear una interfície web, la qual cosa sempre m'ha agradat desenvolupar.

1.2 Propòsit

El propòsit del treball és desenvolupar el portal d'exploració de La Teranyina, similar al que fa etherscan.io sobre Ethereum:

- Ha de mostrar indicadors de les operacions realitzades a la cadena.
- Ha de permetre la cerca de comptes, contractes i transaccions.
- Ha de mostrar el detall de les transaccions.
- Ha de mostrar el balanç i l'historial de transaccions dels comptes i contractes.

Com es podrà comprovar més endavant, la part que té relació amb la màquina virtual d'Ethereum (transaccions d'Ethereum, contractes...) de la Teranyina s'ha descartat per aquest projecte, ja que ens canvia exponencialment l'abast del projecte. El nostre projecte està dedicat a les parts que tenen relació amb Substrate, que també té les seves transaccions i dades. Si haguéssim incorporat la part d'Ethereum en el nostre projecte, estaríem fent quasi dos exploradors en un. Per manca de temps i excés de feina hem decidit retallar aquesta part del projecte final.

1.3 Objectius del projecte

Els objectius d'aquest PFG són els següents:

- Estudiar totes les tecnologies i components en les quals es basa la blockchain Teranyina:
  - Què és una blockchain?
  - Què és Substrate (tecnologia sobre la qual està construïda la Teranyina)?

  - Què és i quins objectius té la blockchain Teranyina?
  - Què són els blocks explorers?

- Desenvolupar una base de dades que contingui els elements que hem de consultar de la Teranyina.
- Desenvolupar una API que ens permeti comunicar i obtenir dades de la base de dades.
- Desenvolupar una interfície web on poder mostrar de forma més amena totes les dades de la blockchain que tenim filtrades a la base de dades i retornades per l'API.

CAPÍTOL 2

Viabilitat

En aquest estudi es fa una valoració de la conveniència del desenvolupament a mida, una estimació dels costos i una anàlisi dels riscos. I, basant-se en tot això, es dictamina la viabilitat del projecte.

2.1 Conveniència del desenvolupament a mida

Tot i que ja existeixen exploradors de propòsit general que ens permetrien consultar la blockchain de La Teranyina, s'opta pel desenvolupament a mida perquè, en un futur, a mesura que La Teranyina evolucioni s'hi introduiran especificitats que un explorador genèric no podria cobrir.

2.2 Riscos

El més rellevant és el profund desconeixement que es té, a priori, de la tecnologia blockchain en general i de la tecnologia sobre la qual s'ha muntat La Teranyina en particular. Afortunadament, hi ha bastants factors que fan que aquest problema no comprometi la viabilitat del projecte:

- Es compta amb el suport de l'equip tècnic que ha desenvolupat La Teranyina al grup de recerca TECNIO Centre EASY, el qual, ja des de les primeres reunions s'ha posat a la meva disposició.
- Hem pogut localitzar una llibreria que ens abstreu de molts dels detalls d'implementació tecnològica de La Teranyina.

2.3 Costos

2.3.1 Recursos humans

Tot i que el desenvolupament del projecte estarà a càrrec de la mateixa persona (jo mateix), hi ha 2 perfils ben diferenciats en funció de les tasques a fer:

Analista. Encarregat de planificar el projecte, documentar-se sobre el problema a resoldre, dissenyar la solució, seleccionar les eines i components a utilitzar en el desenvolupament i fer-ne el control de qualitat.

Programador. Encarregat de desenvolupar la solució i testejar-la.

2.3.2 Recursos materials

Infraestructura software: No es preveu cap despesa en llicències de software ni per les eines de desenvolupament ni per les components de base utilitzades . En tots els casos, es tracta de programari a cost zero.

Infraestructura hardware: Tot el desenvolupament es realitzarà en el PC de la persona encarregada del projecte. Quant a la infraestructura necessària quan es faci el desplegament de la solució, no s'ha imputat cap cost donat que es preveu desplegar-ho inicialment en infraestructura existent.

2.3.3 Cost econòmic

Concepte	**Quantitat**	**Preu**	**Import**
Hores analista	224 hores	17,5 €	2940 €
Hores programador	336 hores	12,5 €	3150 €
Infraestructura software	-	0 €	0
PC desenvolupament	1	125 €	125 €
TOTAL			**6215 €**

Taula 2.1: Cost projecte.

Els preus/hora s'han extret de https://salarios.infojobs.net/.

2.4 Conclusions

Concloem que el desenvolupament a mida és la millor opció donat que:

- Ens permetrà evolucionar el projecte a mida de l'evolució de La Teranyina.
- El risc a causa del desconeixement del problema és assumible gràcies al suport de la gent que ha desenvolupat La Teranyina.
- Els costos són raonables

CAPÍTOL 3

Metodologia

La metodologia emprada pel desenvolupament del projecte ha sigut el **procés en cascada**.

El model de desenvolupament en cascada es caracteritza principalment en desglossar les diferents activitats del projecte en fases seqüencials, on cada fase depèn de la feina feta de l'anterior. És a dir, no es pot començar la següent fase fins que no s'hagi finalitzat la fase actual en la qual s'estigui treballant.

En el model original es destaquen les següents fases, les quals succeeixen en el següent ordre:

1. Determinar i especificar els requisits del projecte a desenvolupar.

2. Dissenyar els diferents elements que permetin donar solució als requeriments.

3. Construir i implementar el disseny (programar la solució).

4. Dur a terme les proves necessàries per determinar el correcte funcionament del nostre programari.

5. Manteniment del programari.

En el desenvolupament de programari informàtic, acostuma a estar entre els enfocaments menys iteratius i flexibles, ja que el progrés flueix en gran part en una direcció (cap avall, com una cascada) a través de les diferents fases esmentades anteriorment.

Hem escollit aquesta metodologia perquè s'adapta a projectes petits amb objectius clars des de l'inici, com és el cas del nostre projecte. Des del principi, ja es tenia molt clar quines components s'havien de desenvolupar i en quin ordre perquè és el patró que segueixen la majoria d'exploradors que trobem a la xarxa.

A més a més, en ser un projecte desenvolupat per una sola persona, hi ha menys riscos de tenir problemes d'integració de les diferents components, els quals, metodologies iteratives com, per exemple, SCRUM intenten mitigar.

Per tant, hem cregut que usant la metodologia en cascada seria la més adient a l'hora de desenvolupar el projecte.

3.1 Control de versions

Per tal d'emmagatzemar el codi en algun lloc que no sigui el nostre propi ordinador i poder portar un control dels canvis fets hem decidit tenir el nostre codi en el Github.

GitHub és una pàgina web i un servei en el núvol que ajuda els desenvolupadors a emmagatzemar i gestionar el seu codi, així com a rastrejar i controlar els canvis en el seu codi. Ens dona una gran flexibilitat, permetent desenvolupar des de qualsevol altre ordinador.

CAPÍTOL 4

Planificació

En aquest capítol es procedirà a descriure els diferents detalls de la planificació del projecte per tal de complir amb els objectius de la introducció.

Seguint la nostra metodologia en cascada, la planificació la podríem dividir en diferents etapes seqüencials.

1. Estudi de les tecnologies en què es basa el nostre projecte.
2. Desenvolupament del projecte.
3. Documentació del projecte.

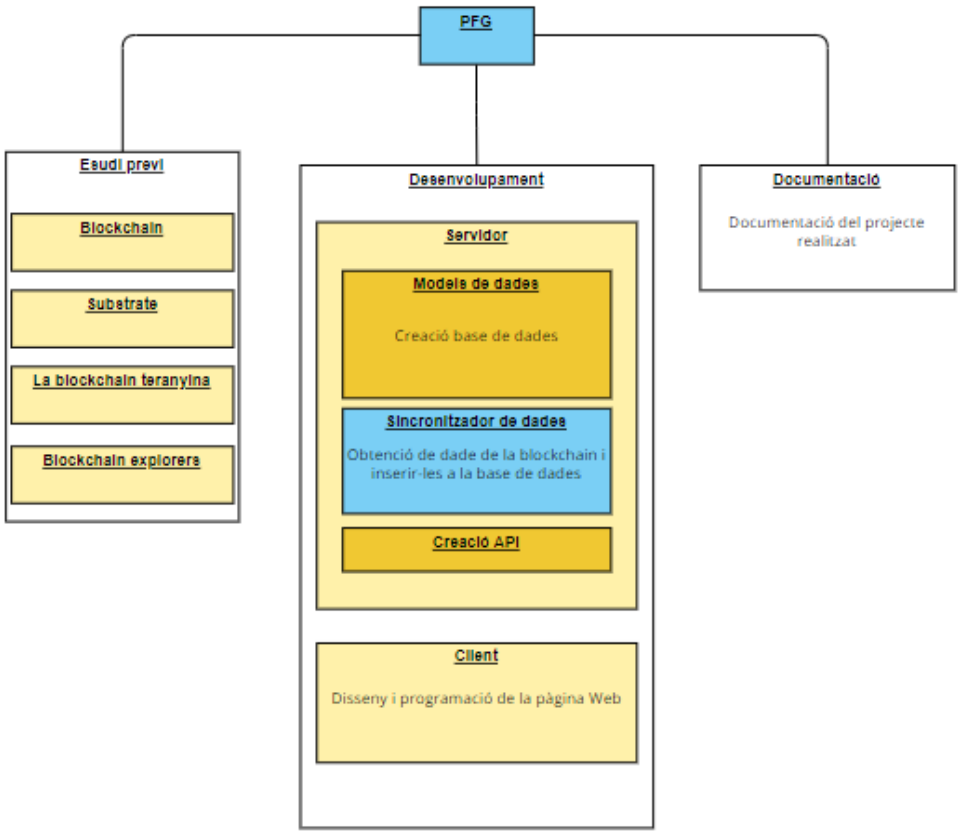

Figura 4.1: Diagrama planificació seqüencial en cascada

Com es pot observar en el diagrama anterior, les etapes són molt clares i definides. Primerament, necessitarem documentar-nos sobre les tecnologies en les quals treballarem: la tecnologia blockchain, Substrate (tecnologia amb la qual es va crear la blockchain Teranyina), la pròpia blockchain teranyina (objectius, estructura...), els block explorers.

Una vegada acabada aquesta primera etapa, procedirem a desenvolupar el projecte en si.

Un dels primers passos que es va determinar va ser la creació de dos repositoris git, un per la part client i l'altre per la part servidor. Es van crear per separat perquè el principi, al tractar-se de desenvolupament en cascada, vam començar desenvolupant la part del servidor (models de dades, sincronitzador de dades, Api) i a posteriori vam desenvolupar la part del client.

Reiteradament, durant el desenvolupament del projecte hem anat incorporant els nous canvis en aquests dos repositoris.

Finalment, una vegada desenvolupat el projecte amb les proves corresponents per comprovar la correctesa de totes les seves parts es procedeix a documentar el projecte seguint l'esquema penjat al Moodle de l'assignatura Projecte Fi de Grau.

4.1 Diagrama de Gantt

Un diagrama de Gantt és una eina de gestió de projectes utilitzada per visualitzar totes les tasques des del començament d'un projecte fins a la seva finalització. A continuació es mostrarà el diagrama de Gantt del nostre projecte:

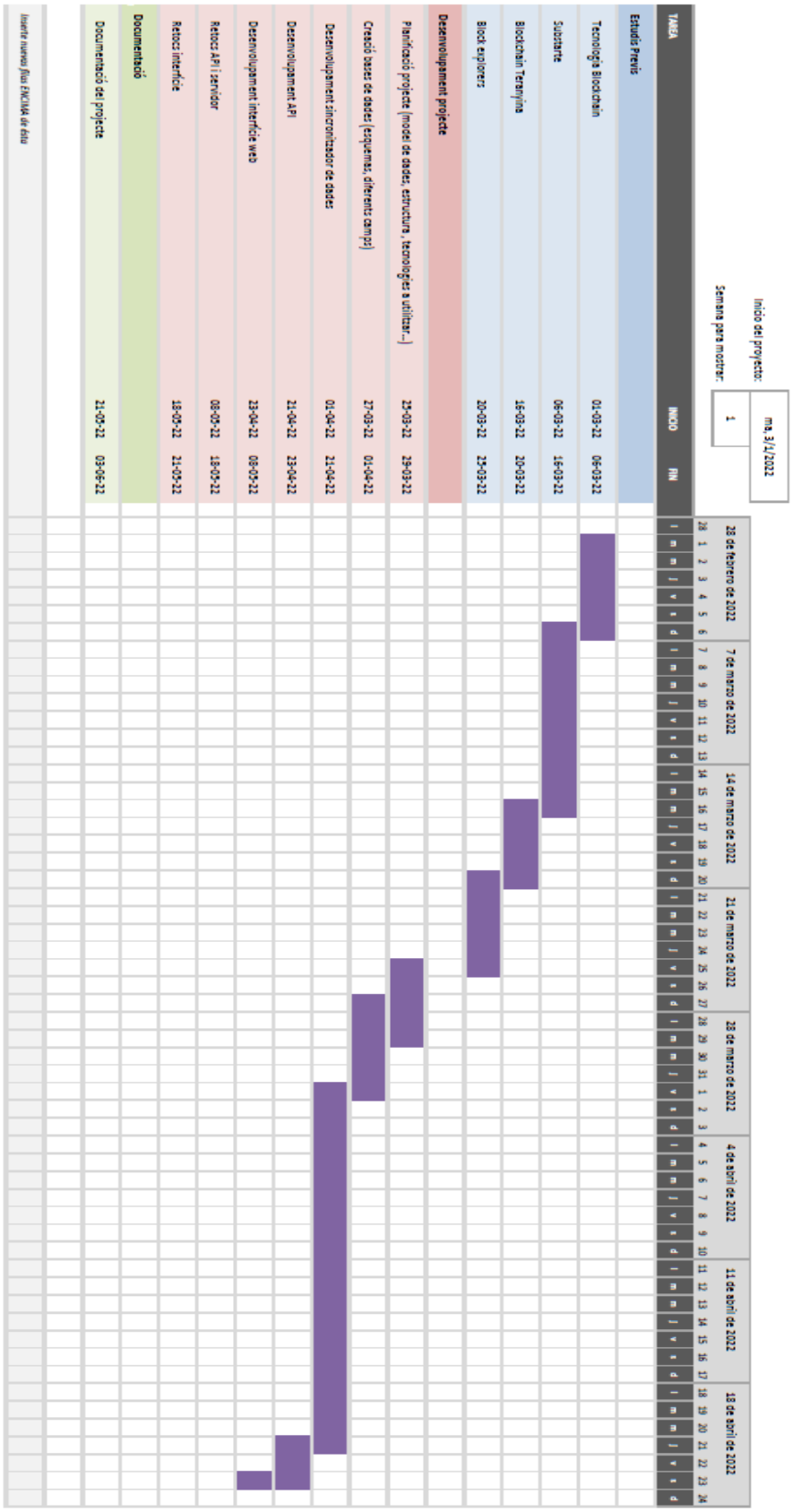

Figura 4.2: Diagrama de Gantt setmana 1 a la 8

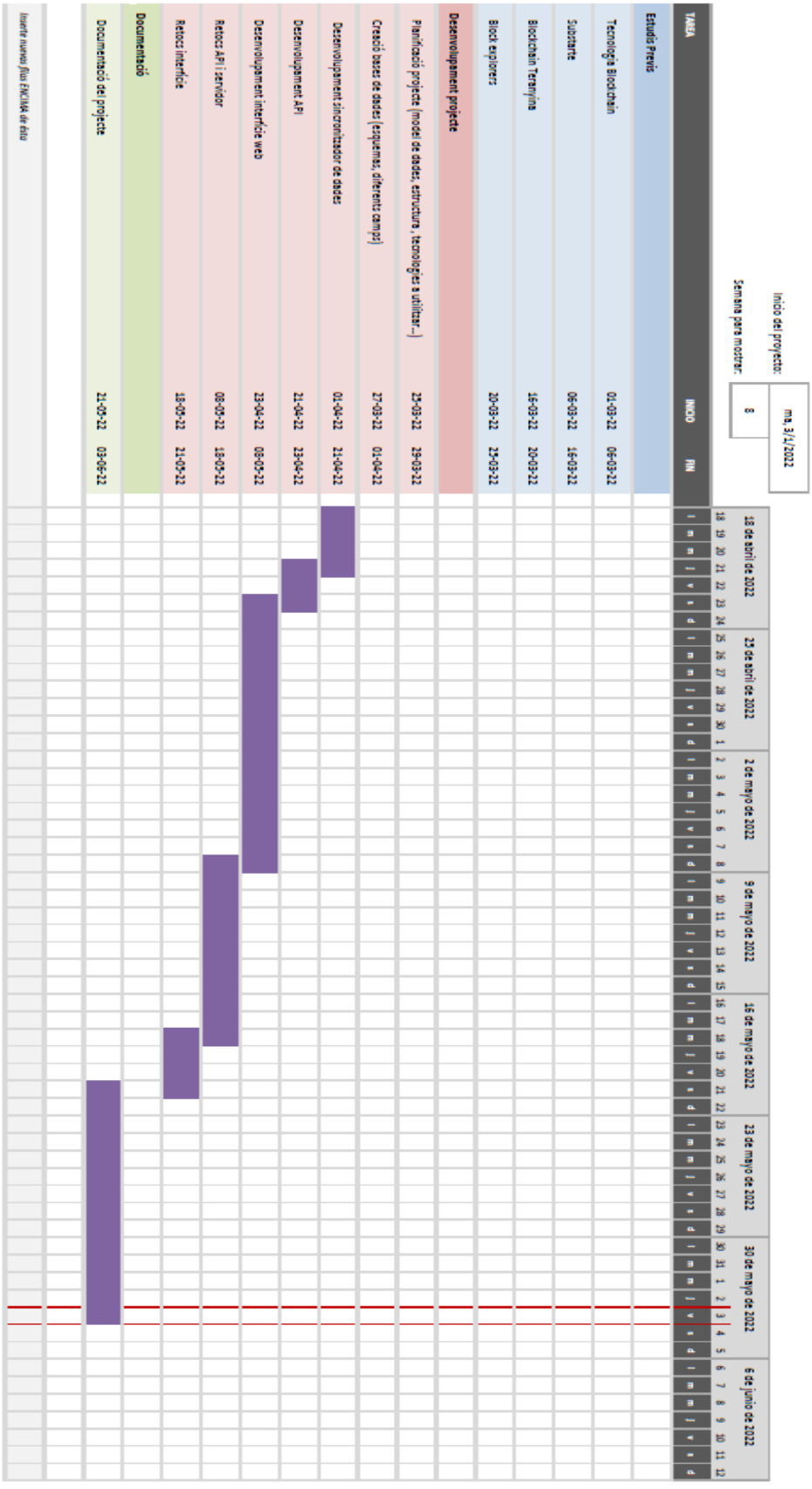

Figura 4.3: Diagrama de Gantt setmana 8 a la 14

CAPÍTOL 5

Marc de treball i conceptes previs

5.1 La tecnologia Blockchain

En poques paraules, una blockchain és una base de dades distribuïda que es comparteix entre els nodes d'una xarxa d'ordinadors. Com a base de dades, una blockchain emmagatzema informació electrònicament en format digital.

De la mateixa manera que una base de dades pot guardar-se tota mena d'informació electrònica, una blockchain també.

Doncs, si totes dues tecnologies serveixen per al mateix propòsit, quines diferències hi ha i perquè en els darrers anys la tecnologia blockchain ha augmentat en popularitat?

Bé, la diferència més clara, com hem comentat abans, és que una blockchain és una base de dades distribuïda, dit d'una altra manera: descentralitzada. En canvi, una base de dades convencional, es troba en un lloc concret, diguem-li servidor, mentre que una blockchain es troba en una xarxa de diferents ordinadors.

5.1.1 Blockchain i el món econòmic

Com hem estat comentant, una blockchain pot emmagatzemar tota mena de dades, però la seva popularitat resideix extensivament en el món empresarial i econòmic. Abans que la tecnologia blockchain arribés al mercat, cada departament d'una organització tenia el seu propi conjunt de dades. Aquestes dades s'enviaven a altres departaments que havien de verificar-les per poder-hi treballar pels seus propòsits.

En aquests entorns és on la tecnologia blockchain ens és útil. Assegura que es mostri la font adequada de dades mantenint la seva validesa, seguretat i qualitat. D'aquesta manera es millora l'accés compartit a les dades que es tradueix en una millor productivitat d'una empresa. Les empreses treballen amb informació (dades). Com més ràpid es rebi i més exacte sigui, millor. La tecnologia blockchain és ideal per lliurar aquesta informació, ja que proporciona informació immediata, compartida i completament transparent emmagatzemada en un llibre de comptabilitat immutable al qual només poden accedir els membres de la xarxa autoritzats.

Una xarxa blockchain pot fer un seguiment de comandes, pagaments, comptes, producció i molt més. I com que els membres comparteixen una única visió de la veritat, poden veure tots els detalls d'una transacció d'extrem a extrem, donant més confiança.

Tot seguit detallem les raons principals per les quals s'aposta en la tecnologia blockchain en el món econòmic.

- **Transparència**: La informació es reparteix entre tots els participants. Tots ells poden veure o vigilar la transacció en curs en el grup. Tot està clarament exposat a la xarxa. Per tant, no hi ha possibilitat de discrepància.

- **Seguretat**: Blockchain utilitza la tecnologia d'encriptació per protegir les dades. Un algoritme criptogràfic que protegeix les dades de qualsevol atac. Cada nou bloc guarda la informació de l'últim bloc i el fa totalment interconnectat entre si.

- **Sense necessitat d'intermediaris**: Una persona pot enviar, rebre o dur a terme qualsevol activitat financer en qüestió de minuts, sense cap necessitat d'intermediaris.

- **Evita estafes de pagament**: Atès que tot es comptabilitza, és difícil que hi hagi discrepàncies o corrupció. Si es produeix una transacció entre dues parts, es necessita la signatura d'ambdues parts evitant qualsevol classe de frau.

5.1.2 Aspectes tècnics de la tecnologia blockchain

En essència una blockchain és com un fitxer que poguéssim tenir a l'escriptori de l'ordinador, però en lloc de representar un document de text o una imatge, el fitxer blockchain representa una història d'intercanvis, transaccions...

Els bancs tenen un arxiu similar a mà en el seu ordinador. Quan els seus clients volen intercanviar diners, obren l'arxiu i afegeixen la transacció proposada. Com a resultat d'afegir a aquest fitxer, els saldos dels clients canvien. Atès que cada banc és propietari del seu fitxer d'historial de transaccions, cada banc pot canviar el seu historial de transaccions a voluntat.

Fer canvis a una blockchain és diferent. Suposem que hem descarregat una còpia d'una blockchain, i ara volem afegir-hi una nova transacció. Obrim el fitxer i escrivim una transacció nova. D'acord, fins aquí hem afegit amb èxit una transacció a la còpia del fitxer blockchain al nostre ordinador, però com s'aconsegueix que tothom que tingui una còpia del fitxer accepti que la nostra nova transacció és vàlida? Les blockchains arriben a un consens no confiant en una sola entitat per declarar que tot és com hauria de ser, sinó més aviat mitjançant

l'ús d'algoritmes de codi obert formats per principis criptogràfics i incentius econòmics per permetre que tothom arribi a la mateixa conclusió. En lloc de confiar en les empreses que busquen beneficis, la confiança es diposita en les matemàtiques i la probabilitat.

5.1.2.1 Blocs

Per fer-ho simple, un bloc en una Blockchain conté una llista de dades (transaccions, informació, diferents registres...), una marca de temps (temps UNIX), un hash criptogràfic de bloc anterior, el hash del bloc actual i un nonce.

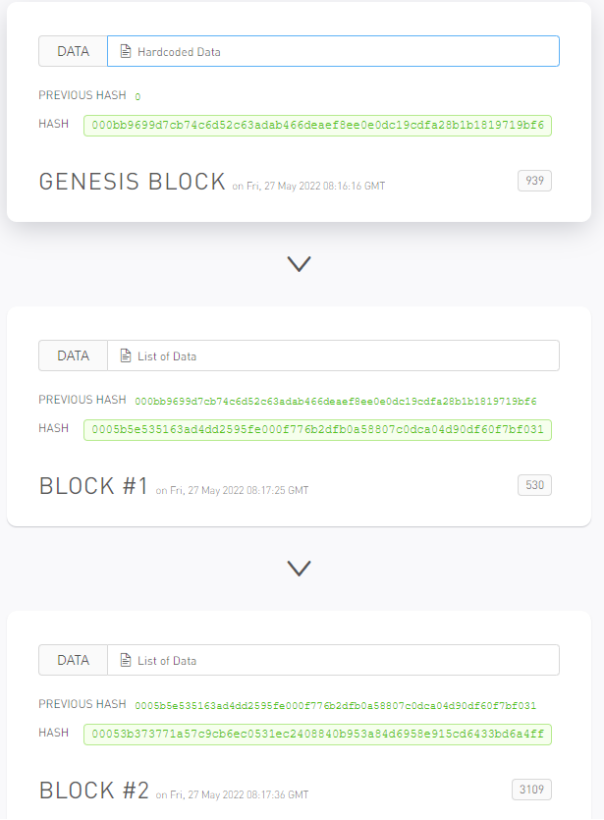

Figura 5.1: Exemple de blocs en una blockchain

- Cada blockhain comença amb un bloc, el bloc 0 o també anomenat Gensesis block.
- El Timestamp és un registre de quan es va crear el bloc. El Timestamp ajuda a mantenir el blockchain en ordre.
- El Hash és un valor alfanumèric que identifica de manera única les dades, o l'"empremta digital"de les dades.

- Un hash vàlid per a un blockchain és un hash que compleix un determinat requisit. Per a aquesta cadena de blocs, tenir tres zeros al principi del hash és el requisit d'un hash vàlid. El nombre de zeros inicials necessaris és la dificultat.

- Una funció de hash pren les dades com a entrada i retorna un hash únic.

  f (dades) = hash

  Atès que el hash és una empremta digital de tot el bloc, les dades són la combinació d'índex, marca de temps, hash anterior, dades de blocs i nonce.

  f (índex + hash anterior + marca horària + dades + nonce) = hash

- El hash anterior és el hash del bloc anterior. El hash anterior del bloc gènesi és "0"perquè no hi ha cap bloc anterior.

- Cada bloc pot emmagatzemar dades en contra. En criptomonedes com Bitcoin, les dades inclourien transaccions de diners.

- El nonce és el número utilitzat per trobar un hash vàlid. Per trobar un hash vàlid, hem de trobar un valor nonce que produeixi un hash vàlid quan s'utilitzi amb la resta de la informació d'aquest bloc.

- El procés de determinació d'aquest nonce s'anomena mineria. Comencem amb un nonce de 0 i continuem incrementant-lo per 1 fins que trobem un hash vàlid.

- A mesura que augmenta la dificultat, el nombre de possibles hashes vàlids disminueix. Amb menys hashes vàlids possibles, es necessita més potència de processament per trobar un hash vàlid.

- Si modifiquem les dades d'algun bloc de la cadena, com que les dades són una variable d'entrada per al hash, canviar les dades canviarà el hash. El nou hash no tindrà tres zeros inicials i, per tant, esdevindrà invàlid. Els blocs posteriors també no seran vàlids. Un canvi d'haixix provocarà una mutació en el hash anterior dels blocs posteriors. Atès que el hash anterior s'utilitza per calcular el hash, els hashes posteriors també canviaran.

  Això comportarà una invalidació en cascada dels blocs.

5.1.3 Tipus de Blockchains

Hi ha 4 tipus de blockchains:

1. **Public blockchain**: Les Blockchain públiques són xarxes obertes les quals qualsevol individu es pot unir i descentralitzades d'ordinadors accessibles per a qualsevol persona que vulgui sol·licitar o validar una transacció. Els usuaris (miners) que validen les transaccions reben recompenses. Les blockchain públiques utilitzen mecanismes de "proof-of-work"o "proof-of-stake". Dos exemples comuns de blockchains públiques inclouen les blockchains de Bitcoin i Ethereum (ETH).

2. **Private blockchain**: Una xarxa blockchain privada és similar a una xarxa blockchain pública. No obstant això, en una privada una organització governa la xarxa, controlant qui pot participar, executar un protocol...

3. **Hybrid Blockchains**: Diverses organitzacions poden compartir les responsabilitats de mantenir un blockchain. Aquestes organitzacions preseleccionades determinen qui pot enviar transaccions o accedir a les dades.

4. **SideChains**: Una Sidechain és una cadena de blocs paral·lela a la cadena principal. Permet als usuaris moure actius digitals entre dues cadenes de blocs diferents i millora l'escalabilitat i l'eficiència.

5.2 Substrate

Substrate és un framewok de codi obert, modular i extensible que serveix per construir blockchains.

5.2.1 Origen

El 2016, el Dr. David Wood, un dels desenvolupadors principals d'Ethereum, va fundar l'empresa Parity. Aquesta empresa ha desenvolupat la xarxa Polkadot, una xarxa de blockchains independents i especialitzades (parachains) que es poden interconnectar entre elles a través d'una cadena principal (relay-chain) i que poden compartir sistema de consens i seguretat. Durant el desenvolupament de Polkadot, es van adonar que:

- Havien de repetir molta de la feina que ja havien fet quan construïen nodes Ethereum i Bitcoin.
- Fins i tot les blockchains més especialitzades tenen moltes components en comú
- Per fer créixer l'ecosistema de Polkadot, necessitaven facilitar el desenvolupament de noves blockchains

I això els va portar al desenvolupament de Substrate.

5.2.2 Components

Substrate proveeix tots els components que ens calen per a construir el node (substrate client) d'una blockchain.

- **Storage**: Encarregat de guardar l'estat de la cadena.
- **Runtime**: Conté la lògica de validació i processament de les transaccions d'un bloc.
- **P2P**: Gestiona la comunicació amb els altres nodes de la xarxa Substrate.
- **Consensus**: Conté la lògica que permet acordar entre els nodes de la xarxa de quin és el seu estat.
- **RPC**: Mòdul que permet la interacció dels usuaris amb el node via crides HTTP/Websockets RPC.
- **Telemetry**: Exposa les mètriques per a monitoratge del node/xarxa. I cadascun d'aquests components es pot configurar i estendre segons les necessitats de la xarxa blockchain que es vulgui construir.

5.2.3 Conceptes clau

5.2.3.1 Runtime

Conté la lògica de negoci de la cadena: com es validen les transaccions, com s'executen i com aquesta execució afecta l'estat de la cadena. En definitiva, conté el conjunt de regles que regeixen els canvis a la cadena de blocs.

Les seves característiques principals són:

- Estructura modular.
- Implementat amb l'estàndard web de màquina virtual Wasm.
- Permet upgrades/modificacions sense haver de fer forks de la chain

S'estructura en mòduls especialitzats anomenats Pallets. Cada pallet encapsula la lògica d'un domini específic expressada en forma d'una llista tipus, una llista de crides, una llista d'events i una llista d'errors possibles.

Substrate proporciona, per una banda, una cinquantena de pallets predefinits i, per l'altra, un framework (FRAME) per a desenvolupar-ne de nous.

I quan configurem els nodes de la nostra blockchain, haurem d'escollir quins Pallets (lògica) que usarà.

5.2.3.2 Extrinsics

Les "transaccions" que emmagatzemem als blocs d'una cadena Susbstrate s'anomenen Extrinsics. Cada extrinsic registra una crida a una funció d'un Pallet amb uns paràmetres determinats. Hi ha 3 tipus d'extrinsics:

- Inherents: Extrinsics que afegeix l'autor del bloc.
- Transaccions signades: El tipus de transacció més habitual. Contenen l'autor de la transacció i l'import de la comissió que rebrà el validador del bloc que inclou la transacció.
- Transaccions no signades

5.2.3.3 Execució dels blocs

Els canvis a l'estat a la cadena es produeixen a conseqüència de l'execució dels blocs. L'execució d'un bloc consta de les següents passes:

- Validació de les transaccions contingudes al bloc
- Inicialització
- Execució dels extrinsics del bloc. El mòdul de la runtime encarregat de l'execució, crida, un a un, els extrinsics continguts al bloc i seguint un ordre de prioritització preestablert.
- Finalització

5.2.3.4 Events

Durant l'execució d'un extrinsic, el Pallet encarregat d'executar-lo pot emetre Events, que és informació que es vol comunicar a l'exterior a conseqüència d'aquesta crida. Aquest, esdeveniment, igual que els Extrinsics també es guarden a l'estat de la blockchain.

5.2.4 Compatiblitat/integració amb altres tecnologies blockchain

5.2.4.1 Polkadot

Una blockchain desenvolupada amb Substrate pot funcionar perfectament sola, però, si es vol, és molt fàcil d'integrar a la xarxa Polkadot com una subxarxa (parachain) sense haver de fer-hi canvis substancials.

5.2.4.2 Smart Contracts Ethereum

El pallet EVM permet executar codi (contractes) preparat per la màquina virtual Ethereum (EVM) a dins d'una blockchain Substrate. Està dissenyat per emular la funcionalitat d'execució dels contractes Ethereum a dintre de la Runtime Substate.

5.3 La Blockchain teranyina

És una blockchain basada en Substrate desenvolupada pel grup de recerca de la Universitat de Girona TECNIO Centre EASY i promoguda pel Centre BlockChain de Catalunya.

5.3.1 Propòsit

La Teranyina permet gaudir d'una plataforma basada en blockchain sobre la qual els investigadors, universitats i centres de recerca poden desenvolupar els seus projectes i fer les proves necessàries amb un cost molt reduït o gratuït gràcies als faucet tokens: criptomonedes que els usuaris obtindran de manera gratuïta, i que els permetran fer transaccions en la xarxa de Teranyina, que tindrien un cost en qualsevol altra blockchain. Així doncs, Teranyina pot actuar com una testnet, per a projectes early stage; és a dir, una xarxa de blockchain on els estudiants i investigadors poden provar les seves aplicacions abans de llançar-les al mercat de forma oficial.

5.3.2 Funcionalitats

La xarxa s'ha desplegat a principis del 2022 amb unes funcionalitats mínimes que s'hauran d'anar ampliant. L'objectiu és que la Teranyina permeti:

- Gestió d'identitat
- Publicació i comunicació amb Smart Contracts Ethereum
- Creació de Tokens (ERC20)
- Creació de NFTs
- Hashing
- Timestamping
- Multisignatura

5.3.3 Tokeneconomia

La Teranyina té 3 tokens diferents, cadascun d'ells amb una funcionalitat específica:

- Ralet: Moneda nativa de la Teranyina. És la que s'utilitzarà en el procés de staking un cop Teranyina evolucioni cap a un sistema de consens de tipus Proof of Stake.
- Ral Fuel: Moneda que s'usa com a comissió de les transaccions.
- Ral: Moneda estable de la Teranyina. És un tipus de moneda estable algorítmica.

5.4 Blockchain explorers

Entrant ja en el nucli del nostre projecte, crear un explorador per a la blockchain teranyina, necessitem explicar i aclarir què són els blockchain explorers ("Exploradors de Blockchains") i quin tipus d'informació guarden.

Bàsicament, un blockchain explorer és un simple cercador que et permet navegar a través de tots els blocs, adreçes, transaccions i els diferents tipus de dades que una blockchain pugui tenir. Aquests tipus d'exploradors no són exploradors generals que poden obtenir informació de qualsevol blockchain, cada blockchain necessita el seu explorador concret. Per exemple, si utilitzem l'explorador d'Ethereum, no ens seria possible explorar la cadena de bitcoin o Polkadot. Per Bitcoin necessitaries l'explorador de Bitcoin i per Polkadot l'explorador de Polkadot.

5.4.1 Funcionament dels exploradors

Com hem comentat en punts anteriors, una blockchain és un simple fitxer on es guarda informació sobre l'historial de totes les transaccions realitzades en aquesta xarxa. Però per al funcionament adequat d'un explorador de blocs, simplement descarregant una còpia completa d'una blockchain no és suficient. Necessitem que es vagi refrescant durant tot el dia per poder obtenir l'última informació dels últims blocs. Per aquest motiu, els exploradors necessiten un node complet, un participant de la xarxa descentralitzada que validi les transaccions, tenint així accés a informació rellevant sobre aquestes. Un node complet es descarrega tota una còpia de la blockchain i es dedica a actualitzar-la constantment.

Un blockchain explorer és senzillament un programari que navega per aquest node amb l'ajut d'alguna llibreria que ho faci possible, recopilant informació i mostrant la informació d'una manera més clara i entenedora per l'usuari.

Des d'un punt de vista tècnic definiríem la funcionalitat global d'un blockchain explorer de la següent forma:

1. Els exploradors blockchain utilitzen una interfície de programació d'aplicacions (API) amb una base de dades, sigui relacional o no relacional (NoSQL), juntament amb un node blockchain per recuperar informació d'una cadena.

2. L'explorador llavors es pot utilitzar per realitzar crides a l'API per obtenir la informació guardada a la base de dades corresponent, utilitzant GraphQL o crides REST.

3. L'explorador té una pàgina web a través de la qual mostra la informació rebuda del servidor i pot interactuar amb els usuaris, permetent així que l'usuari realitzi cerques o obtenir informació dels diferents elements que componen la blockchain.

CAPÍTOL 6

Requisits del sistema

6.1 Funcionals

Bàsicament, ha de ser un explorador dels objectes d'una blockchain basada en Substrate. Per tant, ha de permetre:

- Cercador de blocs
- Llista paginable de blocs
- Informació detallada d'un bloc, inclosos extrinsics i events
- Dashboard amb informació de l'estat de la blockchain: darrers blocs, darreres transaccions, ...
- Llista paginable de transaccions
- Informació detallada de les transaccions, incloent els accounts que hi intervenen i els events
- Llista paginable d'extrinsics
- Informació detallada d'un extrinsic
- Llista paginable d'events
- Informació detallada d'un event
- Cercador d'accounts
- Llista paginable d'accounts
- Informació detallada d'un account
- Llista de transaccions d'un account

6.2 Usabilitat

Ha de ser una aplicació web amb les següents característiques:

- Disseny web responsiu. La interfície s'ha d'adaptar al dispositiu des del qual s'accedeix a l'aplicació.
- Bon temps de resposta.
- Interfície intuïtiva i fàcil d'usar sense necessitat de manual d'usuari
- Accessible dels navegadors moderns

6.3 Tecnològics

La part servidora:

- S'ha de poder desplegar al núvol sense massa modificacions al codi.
- Ha de ser tan independent del sistema operatiu de base com sigui possible, a fi i efecte que, a l'hora de triar proveïdor al núvol, hi hagi més possibilitats de tria. En aquest sentit, seria desitjable que les components del servidor requerides es poguessin instal·lar en servidors Windows, Linux o en containers Docker.

CAPÍTOL 7

Estudi i decisions

7.1 Tecnologies utilitzades en el desenvolupament del servidor

En el desenvolupament de la part servidor s'han utilitzat diferents eines que han ajudat en l'organització i rapidesa en el procés de desenvolupament d'aquesta part del projecte. Per començar cal esmentar el framework que s'ha utilitzat per construir la base de dades i l'api: NestJs.

7.1.1 NestJs

NestJs és un framework per construir aplicacions eficients i escalables de node.js al servidor. Utilitza JavaScript, està construït amb i suporta TypeScript i combina elements de L'OOP (Programació Orientada a Objectes), FP (Programació Funcional) i FRP (Programació Reactiva Funcional). A més a més, Nest fa ús de frameworks robustos de servidor HTTP com és Express. Un dels principals avantatges de Nest respecte altres frameworks de node.js és que proporciona una arquitectura d'aplicacions que permet als desenvolupadors i equips crear aplicacions altament testejables, escalables i fàcilment mantenibles. L'arquitectura està molt inspirada en Angular.

Diferents característiques rellevants de Nest:

- Tot i que NestJs està pensat per codificar les aplicacions amb TypeScript, també és viable programar amb JavaScript. En el nostre cas hem construït l'aplicació servidor amb el llenguatge Typescript.
- En el procés de desenvolupament de l'API, Nest ens permet decidir entre una API REST o una API GraphQL. Nosaltres ens hem decantat per GraphQL, els motius dels quals s'explicaran en punts posteriors.
- Nest permet fàcilment la integració amb qualsevol base de dades SQL o NoSQL. En el nivell més general, connectar Nest a una base de dades és simplement una qüestió de carregar el controlador node.js adequat per a la base de dades. En el nostre cas, s'ha utilitzat una base de dades NoSQL, en concret MongoDB.

A continuació procedirem a definir les eines de Nest que hem utilitzat per desenvolupar la nostra part del servidor del projecte.

7.1.1.1 TypeScript

Definiríem TypeScript com un llenguatge de programació orientat a objectes que és un superconjunt de JavaScript. Dit en paraules senzilles, TypeScript és JavaScript amb altres funcions addicionals.

TypeScript està construït sobre de JavaScript. En primer lloc, s'escriu el codi amb TypeScript, i tot seguit es compila el codi TypeScript en codi JavaScript sense format mitjançant un compilador TypeScript.

En els seus inicis JavaScript estava pensat per ser utilitzat per a fragments curts de codi (snippets) incrustats en una pàgina web. Escriure grans quantitats de codi no hauria sigut habitual. Tot i això, amb el temps, JavaScript es va fer cada vegada més popular, i els desenvolupadors van començar a utilitzar-lo per no només crear fragments de codi curts, sinó que avui dia hi ha desenvolupadors que només utilitzen JavaScript per programar tot el conjunt de la pàgina web.

Per sintetitzar, tenim un llenguatge que va ser dissenyat per a usos ràpids que va créixer fins a convertir-se en una eina per escriure aplicacions amb milions de línies. Cada llenguatge té les seves pròpies peculiaritats, rareses i sorpreses, i l'humil començament de JavaScript fa que en tingui moltes.

En citem algunes:

- JavaScript no admet tipus. Dit això, pot ser que ens trobem amb anormalitats com la següent.

```
2 + 2; //Outputs: 4
"2" + "2"; //Outputs: "22"
"2" + 2; //Outputs: "22"
```

- JavaScript detecta els errors en temps d'execució. És a dir, fins que no el codi no és executat no podem comprovar si en aquell codi hi ha algun error.

Però, quines són aquestes noves funcionalitats de TypeScript que complementen JavaScript?

- **Tipat estàtic**: a JavaScript no existeixen com a tal els tipus, encara que els enumerats es poden simular amb classes senzilles. Mentre que TypeScript és un llenguatge fortament tipat, on es poden crear tipus genèrics o interfícies.

- **Modularització**: TypeScript ofereix un suport directe per a mòduls, mentre que JavaScript ho fa a través d' ECMAScript 6.
- **Tuples**: JavaScript no les suporta, però si TypeScript.
- **Orientació a objectes (OOP)**: la sintaxi de TypeScript per a la programació orientada a objectes és molt similar a la d'altres llenguatges com Java o C#. A més afegeix classes abstractes i modificadors d'accés, entre altres característiques. A JavaScript també es pot programar orientat a objectes, però és una cosa més complexa.
- **Decoradors**: JavaScript no té suport per a decoradors, mentre que TypeScript sí.
- **Errors**: Destaca els errors en el moment del desenvolupament i no en temps d'execució.

Per què utilitzem TypeScript?

- TypeScript és compatible amb totes les biblioteques i frameworks de JavaScript.
- Amb un augment de la complexitat del codi, JavaScript havia de complir els requisits de l'OOP; TypeScript ens proporciona aquesta solució.
- TypeScript ajuda amb un desenvolupament de codi més ràpid, millorant així el rendiment.
- En projectes petits, utilitzar TypeScript és una mica brusc, per la qual cosa potser no val la pena. Ara bé, en projectes mitjans o grans, i sobretot en un equip de desenvolupament, escriure codi en TS ofereix grans avantatges que s'han de notar a curt i a llarg termini.

7.1.1.2 Node.js

Node.js és una plataforma basada en JavaScript molt potent construïda sobre el motor JavaScript V8 de Google Chrome. Node.js és de codi obert, completament gratuït i utilitzat per milers de desenvolupadors de tot el món. npm, l'ecosistema de paquets de Node, és l'ecosistema de biblioteques de codi obert més gran del món.

Les aplicacions web creades amb Node.js es beneficien massivament de la seva capacitat de fer múltiples tasques. A diferència d'altres plataformes, la seva arquitectura d'un sol fil i basada en esdeveniments, processa múltiples sol·licituds concurrents de manera eficient sense obstruir la RAM. A més, les

seves operacions "event-loop" i "non-blocking" d'Entrada/Sortida permeten l'execució de codi a un ritme que afecta significativament el rendiment general de l'aplicació.

De què és capaç Node?

- Node.js pot generar contingut dinàmic de la pàgina.
- Node.js pot crear, obrir, llegir, escriure, suprimir i tancar fitxers al servidor.
- Node.js pot recopilar dades de formulari.
- Node.js pot afegir, suprimir, modificar dades a la base de dades.

Avantatges d'utilitzar Node:

1. Fàcil d'aprendre. Si un desenvolupador coneix bé JavaScript Node.js serà una eina fàcil d'aprendre.
2. Escalable. Node.js té una arquitectura "event-loop" i "non-blocking", que permet als servidors executar-se sense interrupció. Per això, Node.js és un excel·lent framework per al desenvolupament d'aplicacions modernes que poden escalar i reduir-se segons sigui necessari.
3. El codi s'executa ràpidament i això millora tot l'entorn en temps d'execució. Això es deu en gran part al seu sistema de seccions. Però també té a veure amb el fet que funciona amb el motor V8 JavaScript de Google.
4. Api. JavaScript s'utilitza tant en el "front-end" com en el "back-end" de les pàgines. Per tant, un servidor pot comunicar-se fàcilment amb el "front-end" a través d'una API mitjançant Node.js. Node.js també proporciona paquets com Express que fa que sigui encara més fàcil construir aplicacions web.

7.1.1.3 Express

Express és un framework d'aplicacions web node.js flexible que proporciona un conjunt robust de característiques per desenvolupar aplicacions web i mòbils. Facilita el ràpid desenvolupament d'aplicacions web basades en node.

Per què hem d'utilitzar Express juntament amb les nostres aplicacions Node.js?

- Proporciona la gestió de les sol·licituds web amb el mètode de ruta que ofereix.
- Facilita el desenvolupament d'aplicacions de bases de dades MySQL, MongoDB...

- Proporciona un desenvolupament fàcil i ràpid d'aplicacions web.
- Permet configurar middlewares per respondre a les sol·licituds HTTP.
- Permet crear una API per a diferents aplicacions.
- Facilita la gestió d'arxius estàtics.

En el nostre projecte de Nest no ens hem de preocupar de la part d'Express, ja que ens ve incorporada per defecte i ja ens redirigirà al controller o resolver corresponent automàticament.

7.1.1.4 MongoDb

MongoDB és una base de dades NoSQL escalable i flexible basada en col·leccions i documents dissenyada per superar l'enfocament de les bases de dades relacionals.

Si fem memòria, les bases de dades relacionals van néixer a l'era dels mainframes, molt abans que Internet, el núvol, el big data i el telèfon mòbil. Aquestes bases de dades es van dissenyar per executar-se en un sol servidor: com més grans, millor. L'única manera d'augmentar la capacitat d'aquestes bases de dades era actualitzar els servidors (processadors, memòria i emmagatzematge) per escalar.
Però, que és una base de dades NoSQL?

Les bases de dades NoSQL emmagatzemem informació en documents JSON en lloc de columnes i files utilitzades per les bases de dades relacionals. NoSQL significa "not only SQL", la qual cosa implica que una base de dades NoSQL pot emmagatzemar i recuperar dades sense utilitzar SQL, o bé es poden combinar la flexibilitat de JSON amb la potència d'SQL per obtenir el millor dels dos mons. En conseqüència, les bases de dades NoSQL estan construïdes per ser flexibles, escalables i capaces de respondre ràpidament. Existeixen diferents tipus de bases de dades NoSQL: bases de dades de tipus Graph, bases de dades orientades a columnes, magatzems de clau-valor i les bases de dades de documents.

MongoDB és una base de dades basada en documents, el que significa que les dades s'emmagatzemen com a documents i els documents s'agrupen en col·leccions. Si ho haguéssim de traduir a una base de dades SQL, els documents serien les files, mentre que les col·leccions serien les taules. El model de document és molt més natural pels desenvolupadors, ja que els documents són independents i es poden tractar com a objectes. Això vol dir que els desenvolupadors poden centrar-se en les dades que necessiten per emmagatzemar i processar, en lloc de preocupar-se per com dividir les dades en diferents taules i relacions.

En el seu interior els documents poden contenir molts parells de clau-valor diferents, o parells de clau-matriu, o fins i tot documents niats.

Característiques pròpies de MongoDB:

1. **Ad Hoc Queries Support**. Quan estem en la fase de disseny de bases de dades, no tenim ni idea de quines consultes es podrien executar. Per tant, quan diem, MongoDB dona suport a Ad Hoc Queries, vol dir que el MongoDB admet consultes que no es coneixien mentre s'establia una estructura per a la base de dades.

2. **Índexs**. Sense els índexs adequats, una base de dades es veu obligada a escanejar documents un per un per identificar els que coincideixen amb la sentència de la consulta. Però si hi ha un índex adequat per a cada consulta, les sol·licituds de l'usuari poden ser executades de manera òptima pel servidor. MongoDB ofereix una àmplia gamma d'índexs i característiques amb ordres d'ordenació específiques del llenguatge que admeten patrons d'accés complexos als conjunts de dades.

3. **Replicació**. MongoDB proporciona alta disponibilitat i redundància amb l'ajuda de la replicació, crea múltiples còpies de les dades i envia aquestes còpies a un servidor diferent de manera que si falla un servidor, les dades es recuperen d'un altre servidor.

4. **Schema-less**. Això vol dir que hi pot haver diversos documents en una col·lecció, amb claus diferents, i és possible que aquestes claus no es trobin en altres documents. Aquest és el motiu principal de la flexibilitat de les dades en MongoDB.

5. **Sharding**. És a dir, els grans conjunts de dades es divideixen i es comparteixen entre diverses màquines. Les dades massives poden causar problemes inesperats, però la implementació de la fragmentació pot ser útil. La fragmentació és el procés de partició i difusió de bases de dades entre diverses màquines, mentre que la rèplica és el procés de fer diverses còpies de la base de dades. Les dades es distribueixen en diverses col·leccions, i aquestes col·leccions es coneixen com a "Fragments".

Diferències bàsiques entre una base de dades relacional i MongoDB (no relacional):

Relacional	**MongoDB**
No és adequat per a l'emmagatzematge jeràrquic de dades	Adequat per a l'emmagatzematge jeràrquic de dades
És verticalment escalable (increment de la RAM)	Horitzontalment escalable (més servidors)
Té un esquema predefinit	Té un esquema dinàmic
Més lent	Més ràpid
Suporta JOINS complexos	No suporta JOINS complexos

Taula 7.1: Diferències entre bases de dades relacionals i mongoDB.

MongoDB proporciona dos tipus de models de dades: un model de dades incrustat (embedded) i un model de dades normalitzat.

- **Model incrustat**
  - En aquest model, podem tenir incrustades totes les dades relacionades en un sol document, també es coneix com a model de dades denormalized. Els documents incrustats capturen relacions entre dades emmagatzemant dades relacionades en una única estructura de document. Els documents MongoDB permeten incrustar estructures de documents en un camp o matriu dins d'un document. Aquests models de dades desnormalitzats permeten a les aplicacions recuperar i manipular dades relacionades en una única operació de base de dades.

    Aquest tipus de model és el que segueix la filosofia de MongoDB: Les dades que s'accedeix conjuntament s'han d'emmagatzemar juntes. Exemple:

```
Botiga
{
    _id: Objectd,
    nom: string,
    adreça: {
        num: string,
        ciutat: string,
        carrer: string,
    },
    empleats: [
        {
            nom: string,
            càrrec: string,
```

```
            salari: int
        }
    ]

}
```

- **Model normalitzat**

  – El model Normalitzat, a diferència del model incrustat, recupera la idea de les taules i les relacions entre taules de les bases de dades relacionals. No tot es troba en un mateix document, sinó que els documents tenen referències a altres documents. Exemple:

```
Botiga
{
    _id: ObjectId,
    nom: string,
},
Adreça: {
    _id: ObjectId,
    botiga: "ObjectId",
    num: string,
    ciutat: string,
    carrer: string,
},
Empleat: {
    _id: ObjectId,
    botiga: "ObjectId"
    nom: string,
    càrrec: string,
    salari: int
}
```

Per què escollim MongoDb?

- Rapidesa i flexibilitat a l'hora de desenvolupar.
- Possibilitat de poder crear models incrustats, que afavoreixen la rapidesa de l'extracció de dades, evitant JOINS.
- Tenint en compte el nostre volum de dades. MongoDB ens ofereix Índexs i consultes potents per obtenir i afegir dades ràpidament.

7.1.1.5 GraphQL

GraphQL és un llenguatge de consulta i temps d'execució del servidor per a interfícies de programació d'aplicacions (API) que prioritza donar als clients exactament les dades que sol·liciten i no més de les necessàries. Va ser dissenyat per optimitzar les crides REST fent les API ràpides, flexibles i aptes per a desenvolupadors. Només té un únic endpoint i es basa en esquemes. Per aconseguir realitzar les mateixes accions que permet REST, GraphQL usa Query, Mutation i Subscription els quals ens permeten realitzar les accions de Crear, Eliminar, Modificar i Consultar.

Un esquema GraphQL està format per tipus d'objectes, que defineixen quin tipus d'objecte es pot sol·licitar i quins camps té.

GraphQL valida les consultes amb l'esquema. La consulta ha de tenir camps i tipus que existeixen a l'esquema. Si la crida graphql correspon amb l'esquema llavors, s'executa la consulta.

El desenvolupador de l'API adjunta cada camp d'un esquema a una funció anomenada resolver. Durant l'execució, el resolver és cridat per resoldre la query, cridant a la base de dades o on sigui que s'emmagatzemin les dades.

Veiem una consulta GraphQL.

```
{
   Botiges {
        nom
        Adreça {
            ciutat
        }
   }
}
```

Com podem observar, les consultes GraphQL són consultes simples que poden ser enteses per a qualsevol. Si ens fixem en la consulta que tenim més amunt, l'únic que estem demanant són totes les botiges amb el seu nom i la seva adreça amb només el camp ciutat. Tota la resta d'informació que puguin emmagatzemar les taules Botiga i Adreça serà omesa.

Evidentment, Botigues serà una consulta que haurem programat amb antelació en un resolver.

Diferències entre GraphQL i REST:

GraphQL	REST
Només té un endpoint (rutes per accedir a les dades)	Té varis endpoints
Obtenim només les dades necessàries	Obtenim totes les dades de la taula, encara que no les necessitem
Segueix un sistema de tipus per definir l'aspecte de l'API (esquemes).	No hi ha cap concepte d'esquema o sistema de tipus.
Més ràpid	Més lent
Sempre torna com a resposta HTTP 200	Bon maneig dels errors
No fa cahche	Fa cache de les dades ja demanades

Taula 7.2: Diferències entre GraphQl i REST.

Per què utilitzem GraphQL?

- Ens assegura més flexibilitat a l'hora de crear consultes.
- Rapidesa en executar i consultes i retornar dades
- Capacitat d'obtenir les dades que necessitem amb una sola consulta.

7.2 Tecnologies utilitzades en el sincronitzador de dades

7.2.1 Polkadot.js(API)

Per poder obtenir les dades de la blockchain i afegir-les a la nostra base de dades, necessitem alguna llibreria o alguna eina que sàpiga llegir un node de la blockchain i ens retorni les dades més mastegades i llegibles. Aquí entra a l'acció Polkadot.js (API). Una API que permet llegir i extreure dades de qualsevol xarxa basada en Substrate com Polkadot mateix o la nostra xarxa Teranyina usant JavaScript.

Una de les coses més importants a entendre sobre el Polkadot.js és que ens permet escoltar permanentment una cadena a espera de nous blocs. Això ens és molt pràctic perquè podem anar inserint automàticament nous blocs a la nostra base de dades quan acaben de ser inserits a la cadena.

```
// subscribe to all new headers (with extended info)
api.derive.chain.subscribeNewHeads((header) => {
  console.log(`#${header.number}: ${header.author}`);
});
```

7.2.2 Mongoose

Mongoose és una llibreria que ens permet modelar objectes de Mongodb a Node.js.

Mongoose proporciona una increïble quantitat de funcionalitats al voltant de la creació i el treball amb esquemes. Ens permet fer tots els tipus de consultes possibles a la base de dades Mongodb: Obtenir, afegir, actualitzar, i eliminar elements, utilització de les consultes aggregate en mongoDB...

7.3 Tecnologies utilitzades en el desenvolupament del client

De la mateixa manera que pel desenvolupament servidor hem utilitzat un conjunt d'eines i frameworks que ens ajuden a desenvolupar i estructurar el projecte amb més facilitat, a la part client hem fet el mateix.

I en aquest cas, les principals tecnologies emprades han estat:

- **Vite** (compilació i configuració),
- **Vue** (estrucutura i funcionalitat),
- **TailwindCSS** (estil de la interfície)
- **Apollo** (connexió i obtenció de dades del servidor)

7.3.1 Vite

Igual que Vue CLI, Vite és una eina de compilació que proporciona una estructura i un servidor de desenvolupament per a projectes bàsics, siguin de Vue, React, Vanilla o altres.

A diferència de Vue CLI, Vite no es basa en Webpack i disposa d'un servidor de desenvolupament propi que aprofita el mòdul natiu d'ES en el navegador. Aquesta arquitectura fa que Vite tingui un rendiment més elevat que un servidor de Webpack, ja que només carrega els mòduls necessaris en temps d'execució. Vite està construït amb Rollup, que també el fa més ràpid.

A mesura que hem començat a construir aplicacions cada vegada més ambicioses, la quantitat de JavaScript que estem tractant també ha augmentat exponencialment. Com més grans siguin els projectes, ens trobem amb una davallada del rendiment important. Sovint pot trigar uns quants segons, fins i tot minuts. Vite, troba solució a aquest problema. El navegador sol·licitarà qualsevol mòdul JavaScript quan ho necessiti a través d'HTTP i el processarà durant el temps d'execució, augmentant així el rendiment de les nostres aplicacions.

7.3.1.1 Vue

Vue és un framework JavaScript per construir interfícies d'usuari. Es basa en HTML, CSS i JavaScript, i proporciona un model de programació declaratiu i basat en components que ajuden a desenvolupar de manera eficient diferents interfícies d'usuari. Els components ens permeten dividir la interfície d'usuari en peces independents i reutilitzables, i pensar en cada peça de manera aïllada.

A continuació, mostrem l'estructura bàsica d'un document Vue.

```
<script>
</script>

<template>
</template>

<style>
</style>
```

Script ens guarda tot el comportament de la pàgina i les funcionalitats de Vue (funcions, obtenció de dades..), template mantindrà l'estructura HTML i style mantindrà l'estil CSS de la pàgina. Aquesta última part no ens caldrà en el nostre projecte, ja que utilitzem un framework anomenat TailwindCss.

Per què utilitzem Vue?

- **Fàcil d'aprendre**. Utilitza Javascript.
- **Llegibilitat i components.** Com que els components es disposen en diferents fitxers i cada component és només un fitxer, el codi és més senzill d'entendre i llegir. També ens permet reutilitzar aquests components en qualsevol altre component, fent possible la reutilització de codi.
- **Routing**. Vue, a través de la llibreria vue-router, ens ajuda a crear enllaços dinàmics i fàcils a altres pàgines.
- **Popularitat**. És un dels frameworks més populrs del mercat, la qual cosa podem trobar infinitat de llibreries i recursos.

7.3.1.2 TailwindCss

Tailwind CSS és un framework CSS "utility-first" utilitzat per construir ràpidament interfícies d'usuari personalitzades (estils CSS).

Que TailwindCss sigui "utility-first" significa que cada classe és una composició CSS amb diferents atributs. En lloc de crear una classe que aplica una sèrie

de regles diferents com en les fulles d'estil convencionals, cada classe conté una o diverses regles específiques.

Tailwind CSS escaneja tots els fitxers HTML, components JavaScript i qualsevol altra plantilla, cercant els noms de classe, generant els estils corresponents i escrivint-los en un fitxer CSS estàtic. És a dir, després d'haver instal·lat i configurat Tailwinnd a la nostra aplicació, només cal que modifiquem el nostre html afegint els noms de classe ja predefinits per TailwindCss per canviar l'estil dels nostres components.

```
<h1 class="text-xl p-12">Hello World</h1>
```

L'exemple anterior, ens augmentaria la mida del text i ens afegiria un padding de 12 a l'element.

7.3.1.3 Apollo

Apollo és una plataforma que ajuda a gestionar el flux de dades entre els clients de l'aplicació (com ara aplicacions web) i el servidor.

Resumidament, Apollo és un conjunt d'eines per ajudar a utilitzar GraphQL a les aplicacions Web i de servidor. En el nostre cas utilitzarem Apollo a la part client, en concret vue-apollo, que és l'extensió de Apollo que pot ser utilitzada dins un projecte Vue.

Les crides per obtenir dades del servidor seràn crides GraphQL. Aquí, no tenim ni POST, ni UPDATE, ni DELETE, sinó query, mutation i subscription.

7.3.1.4 Llibreries vàries

A més de totes aquestes eines hem utilitzat algunes llibreries open Source per millorar l'estètica de la nostra pàgina:

- Jazzicon: Una llibreria que ens permet crear identicons úniques passant-li un hash. L'utilitzarem per crear les icones de les diferents accounts sense que es repeteixi cap patró.
- Heroicons: Una llibreria pensada per utilitzar juntament amb Tailwindcss que ens dona accés a una gran varietat d'icones SVG les quals podem utilitzar per decorar components de la nostra pàgina.

CAPÍTOL 8

Anàlisi i disseny del sistema

L'objectiu del projecte és desenvolupar un explorador per a la blockchain Teranyina. Si repassem quines peces necessitem per crear un blockchain explorer, ens sorgeixen les següents necessitats.

- Necessitem tenir accés a un node de la nostra blockchain per poder obtenir els blocs amb les dades corresponents.
- Necessitem una base de dades a on emmagatzemar totes les dades provinents d'aquest node que forma part de la xarxa de la blockchain.
- Necessitem una API que ens permeti obtenir aquestes dades de la base de dades a través de consultes d'una manera fàcil i eficient segons les necessitats d'un client.
- Necessitem una interfície client que ens permeti visualitzar aquestes dades recollides per l'API de manera visual i estructurada (una pàgina web).

Dit això, el nostre projecte el podríem dividir en 3 parts clares i diferenciades: El client, el servidor i el sincronitzador de dades, que escoltarà a un node de la blockchain i inserirà dades a la nostra base de dades.

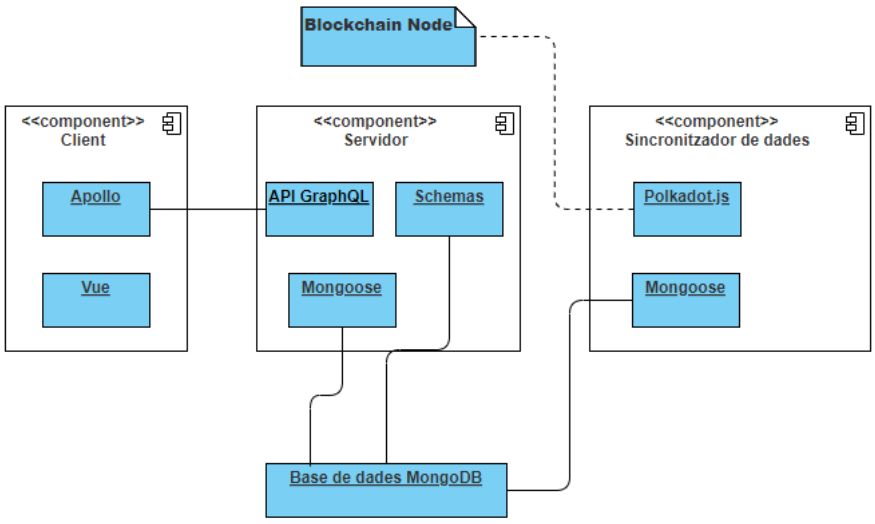

Figura 8.1: Diagrama estructura projecte.

- El client, a través de Apollo, farà crides a l'API de GraphQL per obtenir dades, mentre que a través de Vue crearem diferents vistes per poder mostrar d'una forma més amena les respostes del servidor a l'usuari.

- El servidor mantindrà els esquemes de GraphQL i de MongoDb juntament amb un conjunt de resolvers que formaran l'API de GraphQL. La nostra aplicació serà code first, per tant, la creació dels esquemes MongoDB es mantindran aquí i els nostres resolvers es comunicaran amb MongoDb a través de la llibreria mongoose de Node.js.

- El sincronitzador de dades farà consultes al node de blockchain corresponent, amb l'ajuda de Polkadot.js, obtenint així les dades que després afegirem als documents corresponents de MongoDB a través de mongoose.

Primerament, hem de saber quines dades necessitarem extreure del mateix node de la blockchain, per tant, és essencial que primer construïm el nostre model de dades que guardarem a MongoDB en forma de col·leccions.

Les dades que un blockchain explorer necessita són totes aquelles dades relacionades amb els blocs.

- La nostra primera col·lecció de MongoDB seran els blocs. Cada bloc està format per una capçalera i una sèrie de dades. La capçalera ens dona informació general sobre el bloc: quan ha estat afegit a la cadena, qui és l'autor, els diferents hash...

- La sèrie de dades que emmagatzema un bloc, en el cas d'una blockchain construïda amb Substrate com la nostra, serà una sèrie formada per exrinsics. La nostra segona col·lecció seran els extrinsics.

- Tots aquests extrinsics en essència són un conjunt d'events que llancen els diferents pallets a través de la xarxa. Per exemple, si tinguéssim un extrinsic de transacció (pallet de balances), el mateix pallet llançaria informació sobre les diferents accounts que intervenen en la transacció, la quantitat transferida, la quota a pagar, si la transacció és vàlida o no... Per tant, tindrem una col·lecció on guardarem els Events.

- Com bé sabem, aquests extrinsics poden ser tota mena de dades definides pels diferents pallets que pugui tenir la nostra blockchain. A l'usuari, però, li interessa especialment un tipus d'extrinsic concret: les transaccions. Per tant, és oportú que ens guardem les transaccions i tota la seva informació a banda. Les transaccions tindran una estructura similar als extrinsics.

- Per acabar, necessitem guardar-nos també tota la informació relacionada amb les accounts, ja que ens interessa saber quin és el seu balanç i quines transaccions ha dut a terme. La informació de les accounts no la trobem dins els blocs. Les accounts estan guardades a la metadata de la pròpia blockchain. El que sí que podem trobar dins un bloc són events de creació d'accounts.

Resumidament, les nostres col·leccions seran: blocks, extrinsics, events, transfers i accounts.

Però, com es relacionaran entre elles?

Els "blocks"guarden un llistat d'"extrinsics", aquests extrinsics poden ser de tipus transfer i al mateix temps guarden un llistat d'"events". Paral·lelament, tenim les "accounts".

Com s'ha declarat en el Capítol 6, la nostra base de dades és mongoDB (NoSql). Consegüentment, les nostres col·leccions no seguiran el model relacional convencional, sinó que es basaran en documents i col·leccions.

Nosaltres en el cas dels blocs, els extrinsics i els events som afins al model incrustat que ens proporciona mongoDb. Per tant, els blocs contindran tots els seus extrinsics i els seus events en tota la seva totalitat (amb tots els camps). Seguint aquesta metodologia i indexant les col·leccions per número de bloc, ens serà molt més ràpid obtenir els blocs i les seves dades, i tot amb una sola query. Clar que mongoDb no es guarda tots els extrinsics i events amb tots els seus camps a un document de block, sinó que s'apunta una referència del camp id que té aquell Extrinsic o Event i llavors a l'hora de recuperar les dades d'un bloc, com que l'id està indexat ens retorna les dades dels Extrinsics i els Events amb molta rapidesa.

Aquests Extrinsics i Events, doncs, hauran d'estar a les respectives col·leccions de Extrinsics i Events amb un id que mongoDB genera automàticament.

Els blocs no serà l'única col·lecció amb dades incrustades, Extrinsics i Transfers també tindran els seus Events incrustats de la mateixa manera.

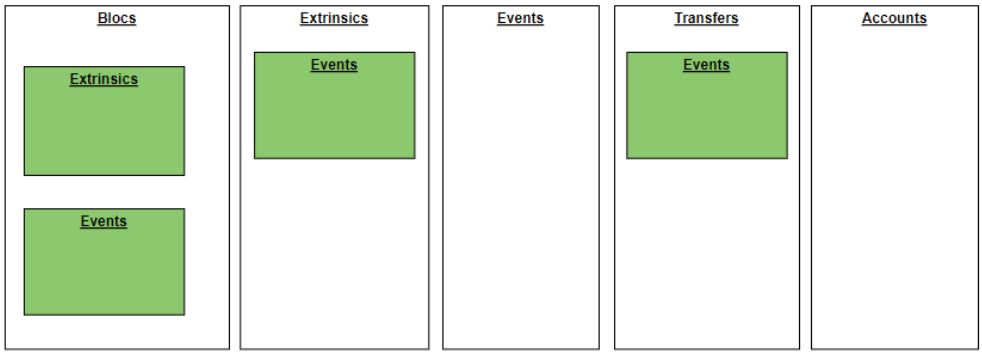

Figura 8.2: Relacions col·leccions base de dades

Tot seguit, procedim a llistar tots els esquemas amb els camps corresponents:

- Block

Figura 8.3: Esquema block

- Extrinsic

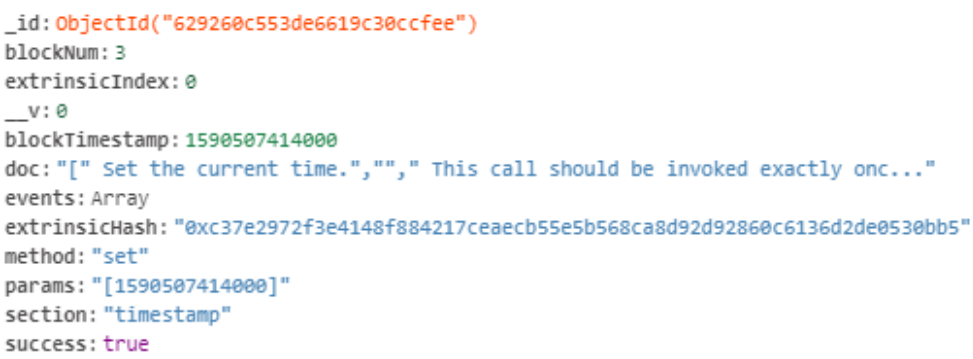

Figura 8.4: Esquema extrinsic

- Event

```
_id: ObjectId("629260c653de6619c30cd00a")
blockNum: 5
eventIndex: 0
__v: 0
data: "[{"weight":158000000,"class":"Mandatory","paysFee":"Yes"}]"
doc: "[" An extrinsic completed successfully."]"
extrinsicIndex: 0
method: "ExtrinsicSuccess"
phase: "{"applyExtrinsic":0}"
section: "system"
```


Figura 8.5: Esquema event

- Transfer


```
_id: ObjectId("629260cd53de6619c30cd0e5")
hash: "0xc8a91d3e028b14086a5ddda5b9e2331d1204fcd94a8182ae5a7c925f6864b1d0"
__v: 0
amount: 2198832999984
blockNum: 10503328
blockTimestamp: 1653760200012
destination: "12xtAYsRUrmbniiWQqJtECiBQrMn8AypQcXhnQAc6RB6XkLW"
events: Array
fee: 157000016
method: "transfer"
section: "balances"
success: true
```


Figura 8.6: Esquema transfer

- Account


```
_id: ObjectId("6293b0ab53de6619c30ce04d")
accountId: "124CHxSDHVTrWKyeeWBqMcS72f2pgoxGjW1qLk5vjwrmeYfd"
__v: 0
availableBalance: "17327999806"
freeBalance: "17327999806"
lockedBalance: "0"
nonce: 16
reservedBalance: "0"
totalBalance: "17327999806"
```


Figura 8.7: Esquema account

Una vegada definides les diferents col·leccions on guardarem totes les dades de la blockchain, podem procedir a definir els 3 principals components que formen el nostre projecte.

8.1 Servidor

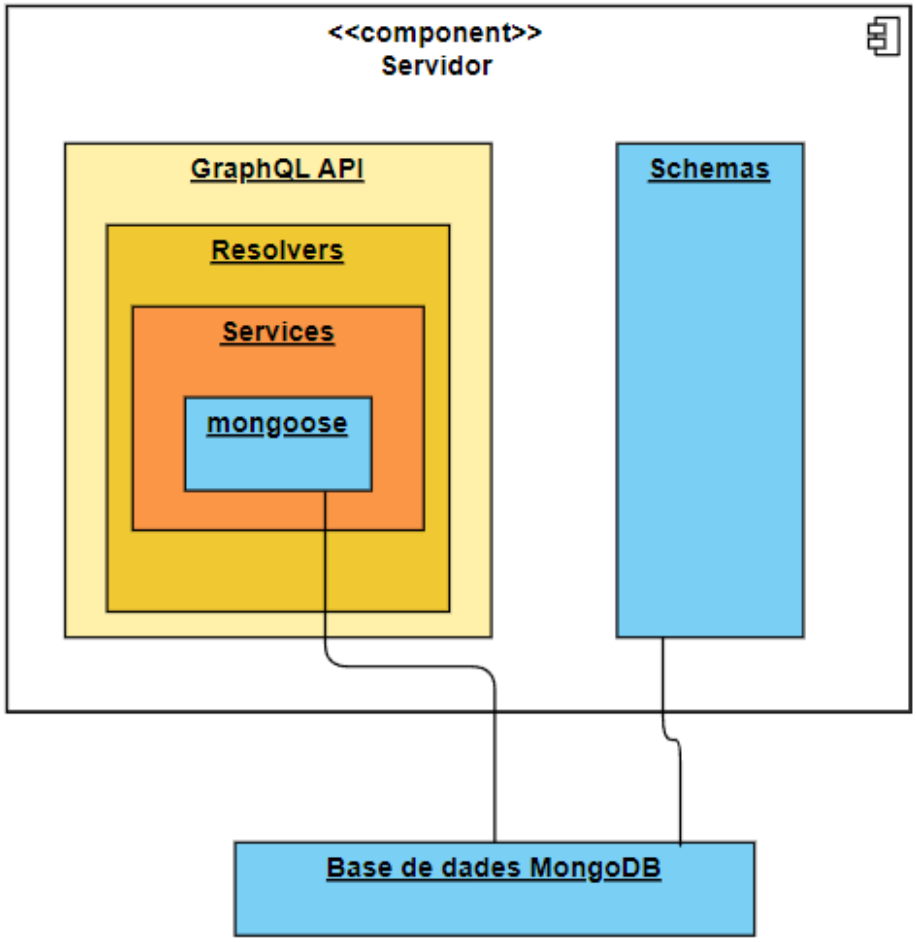

Figura 8.8: Detall servidor

La part servidor té dues funcionalitats bàsiques, encarregar-se de mantenir els diferents esquemes de MongoDb i graphQL i mantenir l'API de GraphQL.

Els esquemes de mongoDB són els explicats en el punt anterior: blocs, extrinsics, transfers, events i accounts. De fet, serà en el mateix servidor on els declararem i els manipularem a gust. Així mateix, els esquemes de GraphQl seran exactament els mateixos.

A l'hora de construir l'API de GraphQL quins tipus de resolvers puguem necessitar en el futur quan construïm la interfície web (Client)?

Ens trobem en el següent escenari:

- Tindrem un sincronitzador de dades, el qual ens carregarà totes les dades provinents del node directament a la base de dades, de manera que no ens caldran consultes d'afegir, actualitzar o eliminar (mutations a graphQL).
- A la part client necessitem que algunes dades es mostrin paginades, és a dir, hem de ser capaços de tornar un conjunt de dades d'una col·lecció dins uns límits que el mateix client ens establirà.
- En el client tindrem diferents finestres mostrant els detalls d'un document d'una col·lecció concreta, per tant, també ha de ser possible que l'API ens retorni documents individuals.
- Ens agradaria poder mostrar algun gràfic i alguns totals a la interfície client, així doncs necessitarem obtenir el nombre de documents que té una col·lecció en un moment concret i obtenir les dades necessàries per poder construir algun gràfic.

Disseny de la API:

- Cada una de les col·leccions mantindrà 3 tipus de queries bàsiques: findAll, findOne i un count, que ens indicarà el nombre d'elements que tenim de la col·lecció.
- Tindrem alguna col·lecció amb alguna consulta aggreggate de mongo per obtenir dades que ens seran útils quan creem algun gràfic.

Tots els resolvers de graphQl seran similars, vegem el resolver de blocs per exemple.

- Resolver

```
1  @Resolver()
2  export class BlockResolver {
3      constructor(private readonly blockService:
       ↪  BlockService) {}
4
5      @Query(returns => [BlockType])
6      async blocks(@Args() blocksArgs: BlockArgs) {
7          return this.blockService.findAll(blocksArgs);
8      }
```

```

    @Query(returns => BlockType, { nullable: true })
    async block(@Args('blockNum') blockNum: Number) {
        return this.blockService.findOne(blockNum);
    }
    @Query(returns => Int)
    async blocksCount() {
        return this.blockService.count();
    }

}
```

- Service

```
@Injectable()
export class BlockService {
    constructor(@InjectModel(Block.name) private
     ↪  blockModel: Model<BlockDocument>) {}

    public async findAll(blockArgs: BlockArgs):
     ↪  Promise<Block[]>
    {
        return this.blockModel.find().sort({blockNum:
         ↪  -1}).skip(blockArgs.skip).limit(blockArgs.take)
                .exec();
    }

    public async findOne(num: Number): Promise<Block>
    {
        return this.blockModel.findOne({blockNum: num})
                    .populate('extrinsics')
                    .populate('events').exec();
    }


    public async count(): Promise<Number>
    {
        return this.blockModel.count().exec();
    }

}
```

Com es pot observar, en realitat el resolver està compost per dues parts: el resolver en sí, que tracta amb tipus de GraphQL (esquema de graphQL) i un servei, que tracta amb les col·leccions de mongoDB i fa les consultes a través de mongoose.

Cal esmentar, també, que en el cas de les Accounts tindrem una consulta que ens recollirà totes les transaccions en què ha participat l'Account. En aquest cas, no tenim les dades incrustades en les accounts, ja que no és com el bloc que està compost d'Extrinsics i aquests Extrinsics després estan compostos d'Events. En el cas de les Accounts, les transaccions no defineixen a les accounts, per tant, no en formen part. Una account pot haver participat en alguna transacció o no.

8.2 Sincronitzador de dades

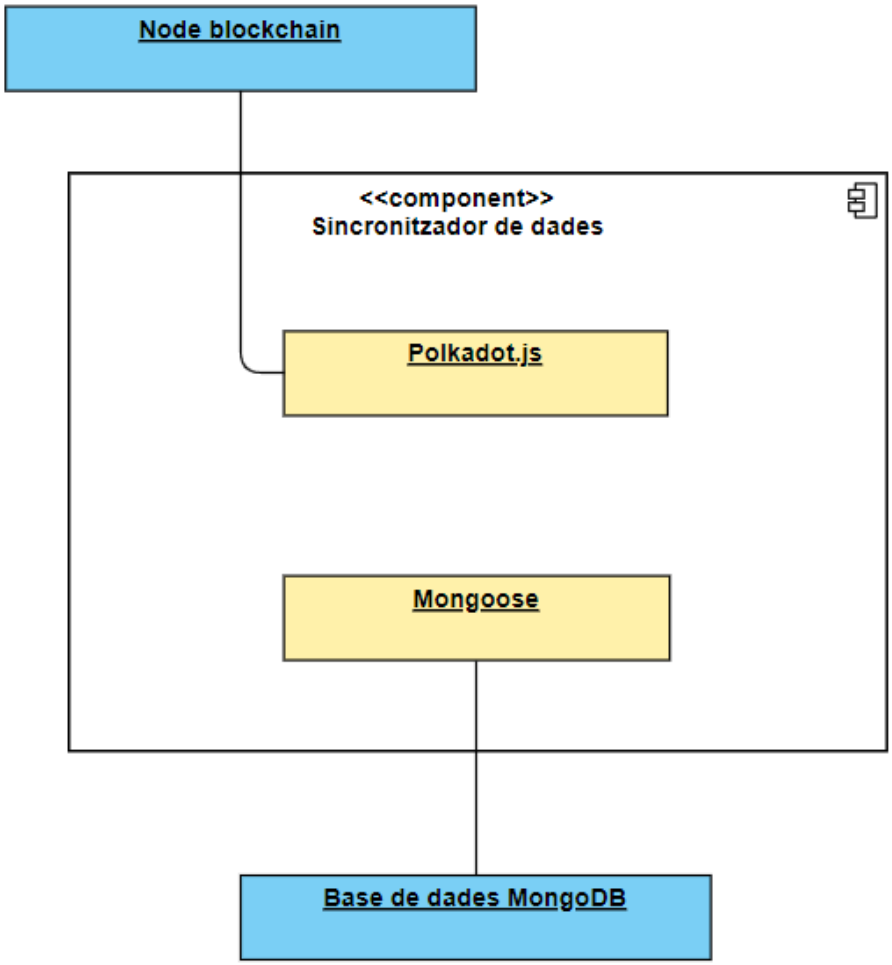

Figura 8.9: Detall Sincronitzador de dades

Com hem comentat a l'inici d'aquest capítol, el sincronitzador de dades obtindrà les dades d'un node i a continuació les afegirà al document corresponent utilitzant Polkadot.js i mongoose respectivament.

Així mateix, ens trobem en el següent escenari:

- Tenim una llibreria, anomenada Polkadot.js, que ens permetrà demanar i obtenir dades d'un node blockchain. crearem els diferents documents amb aquestes dades i inserirem els documents a la col·lecció corresponent, tot això mitjançant mongoose.

- Quan iniciem el sincronitzador, necessitem engegar 3 processos:

    1. Un procés que ens vagi escoltant el node per si algun nou bloc ha estat afegit a la cadena, tractar-lo i inserir-lo a la col·lecció de blocs.

    2. Un procés que ens vagi inserint tots els blocs que ja conté la cadena.

       Atès que podria ser que en algun moment paréssim el sincronitzador, necessitem una funció que ens busqui els rangs de blocs que hem d'afegir, ja que si no començaríem des del bloc zero cada vegada que iniciéssim el procés. Per exemple: Si nosaltres una primera vegada que engeguem el sincronitzador hem aconseguit afegir fins al bloc 1000 i després el parem. La segona vegada que iniciem el procés, aquesta funció ens retornaria el rang corresponent que hem d'afegir, del 1001 fins a l'últim.

       Com que també tindrem un procés que escoltarà els últims blocs, també anirem afegint aquests blocs a la base de dades, per tant, quan parem el sincronitzador podria ser que s'haguessin inserit nous blocs a la cadena sense nosaltres saber-ho. Aquesta funció també ens hauria de tornar aquest rang buit de blocs.

       Recuperant l'exemple anterior, ens tornaria dos rangs a afegir: un primer rang del 1000 al primer que vam començar a escoltar la primera vegada que vam iniciar el procés i un segon rang del bloc que vam parar d'escoltar fins a l'últim que tenim a la cadena.

    3. Un procés que ens vagi inserint totes les diferents accounts que conté el sistema. En aquest cas, no ens cal fer com el punt anterior i només inserir les accounts que no tenim, ja que el procés d'obtenir i inserir totes les accounts és molt ràpid.

Dit això, el nostre disseny serà el següent:

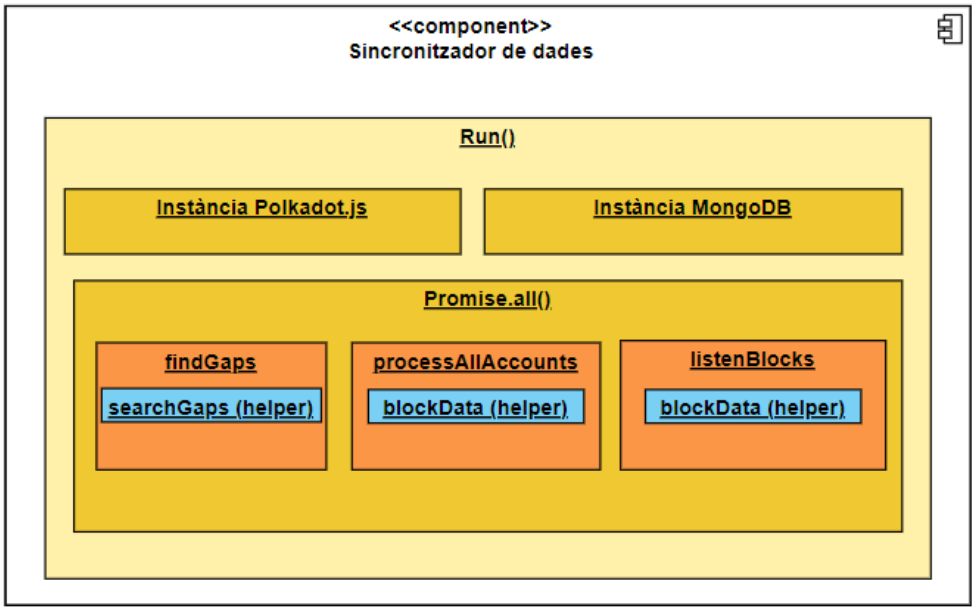

Figura 8.10: Disseny sincronitzador de dades

- Tindrem una funció principal "Run()"que contindrà les següents funcionalitats:

    1. Connectar amb el node de la blockchain Teranyina, a través de Polkadot.js.

    2. Connectar amb la base de dades MongoDB a través de mongoose.

    3. Llançar en paral·lel els processos d'escoltar la cadena pels nous blocs afegits (listenBlocks), afegir els blocs que ens falten per inserir de la cadena (findGaps) i afegir totes les accounts (processAllAccounts).

- Tindrem diferents Helpers, un primer que ens ajudaran a processar les dades d'un bloc: extrinics, transfers, events i accounts (la creació de noves accounts queda registrada a un extrinsic del bloc). I un segon que ens trobarà els rangs dels blocs que ens falten inserir usant una cerca dicotòmica.

8.3 Client

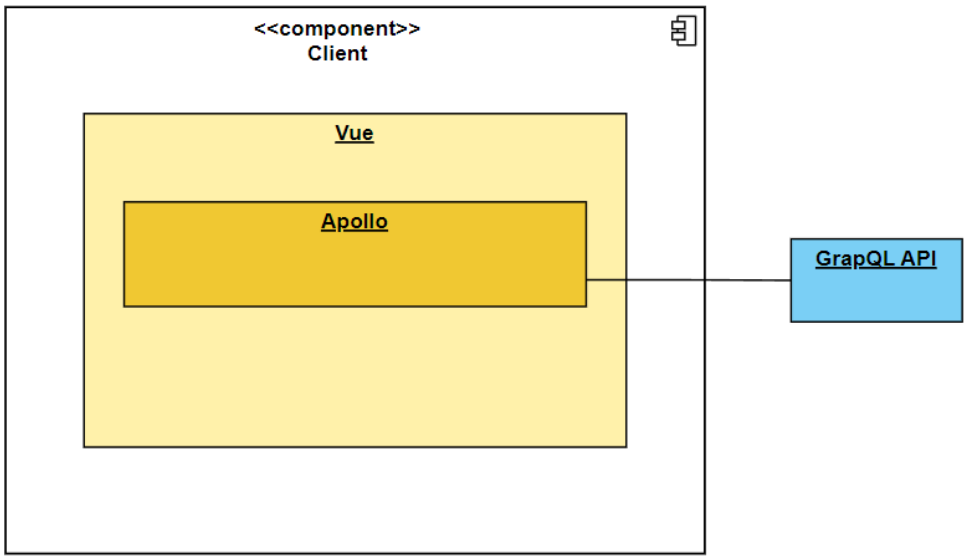

Figura 8.11: Detall client

El client utilitzarà Apollo per comunicar-se amb l'API de GraphQL i mostrar els resultats mitjançant una pàgina web.

En els punts anteriors, hem destacat i explicat quina estructura de dades utilitzarem, com està construïda la nostra API i com carregarem les nostres dades a la base de dades. En aquesta part client necessitem analitzar com organitzar i mostrar totes aquelles dades provinents de l'API.

Observant altres exploradors de Blockchain com etherscan, polkastats o subscan, per mencionar-ne alguns, ens adonem que la informació que mostren els clients web són molt similars.

- Tots tenen una barra de navegació amb el menú principal i els continguts de la blockchain (blocks, transfers...).
- Es manté una barra de cerca a totes les pantalles, per poder buscar blocs, transaccions o accounts.
- A totes les pantalles es manté sempre la barra de navegació a la part superior i just a sota la barra de cerca.
- A la pantalla d'inici es mostra informació general de la cadena (varia depenent de l'explorador) i els últims blocs i transaccions que han estat registrats a la cadena.

- Tenim una finestra amb una llista de blocs paginada, ordenada del més recent al més antic. Aquesta llista només ens mostra la informació més rellevant sobre els blocs. Per obtenir més detalls sobre el bloc, cada bloc té una pàgina associada amb tots els seus detalls, incloent-hi els extrinsics i els diferents events que s'han llençat.

- Aquesta llista amb la seva finestra de detalls també existeix per cada un dels elements del bloc: extrinsic, transfer, events. I per les Accounts.

El disseny final de la nostra interfície és el següent:

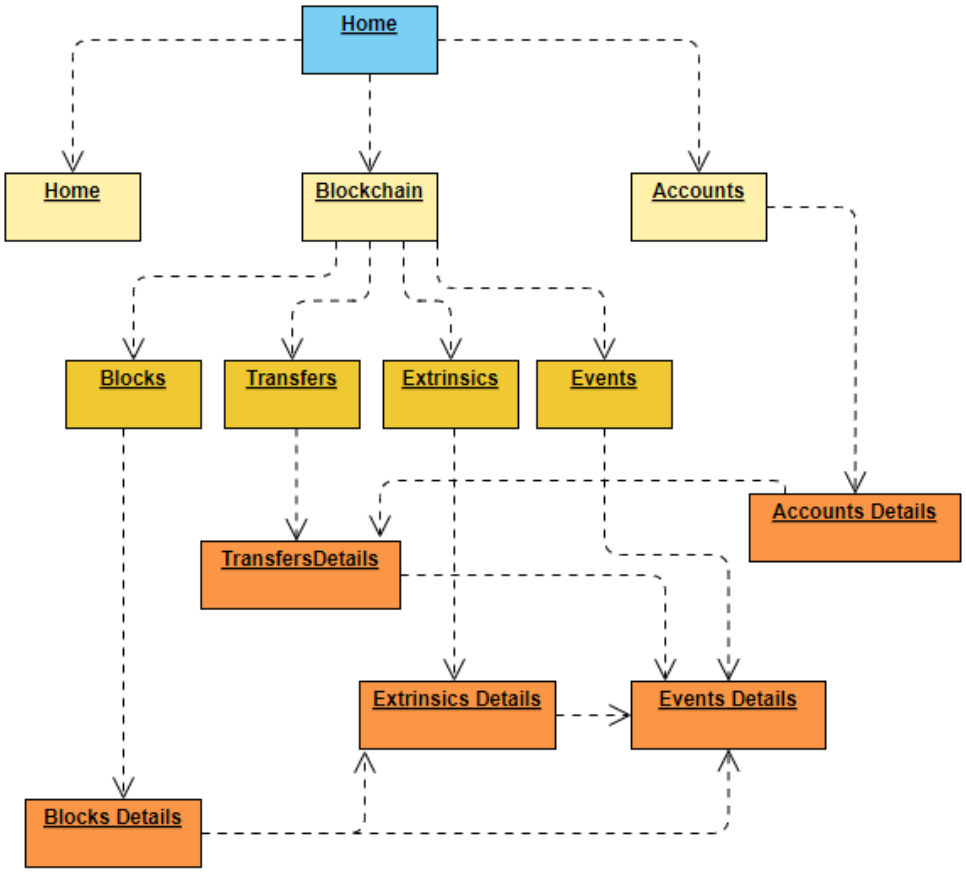

Figura 8.12: Disseny Web

Com es pot observar, els detalls també tenen accés entre si, ja que si recordem, els extrinsics tenien events i les accounts a les transaccions.

8.3.1 Home

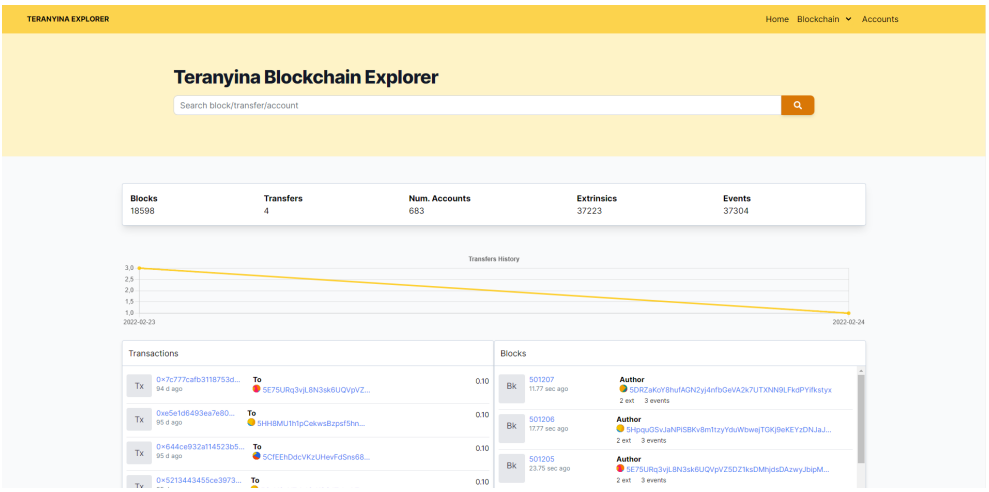

Figura 8.13: Finestra Home

La pàgina principal (Home) està dividida en diferents parts:

- Un títol amb el cercador.
- Un requadre amb informació general de la cadena. En aquest cas són comptadors dels elements que tenim.
- Un gràfic amb el nombre de transaccions històric. Està configurat per dia, ja que a la teranyina no tindrem gaires transaccions. En un futur, es podria canviar per mostrar només les transaccions mensuals.
- Dues llistes de les últimes 10 transaccions i els últims 10 blocs, amb la informació més rellevant.

8.3.2 Llistes

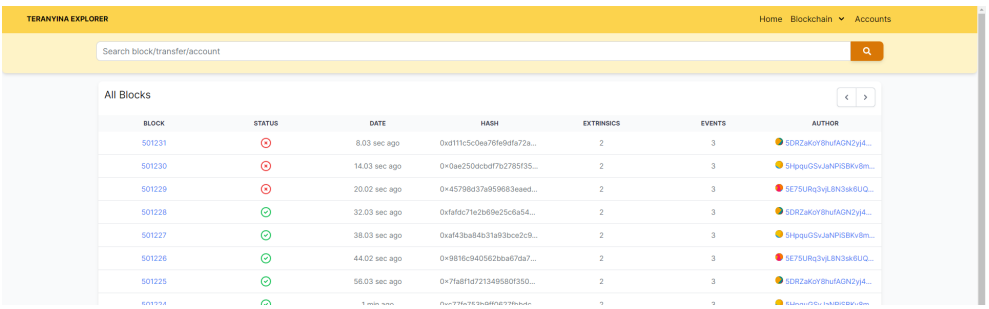

Figura 8.14: Finestra Llista blocs

Cada una de les pàgines que mostren una llista (blocks, Transfers, Extrinsics, Events) està construïda de la mateixa manera.

- Un títol que indica quina col·lecció estem llistant
- Una simple paginació on els elements es mostren de 25 en 25.

8.3.3 Detalls

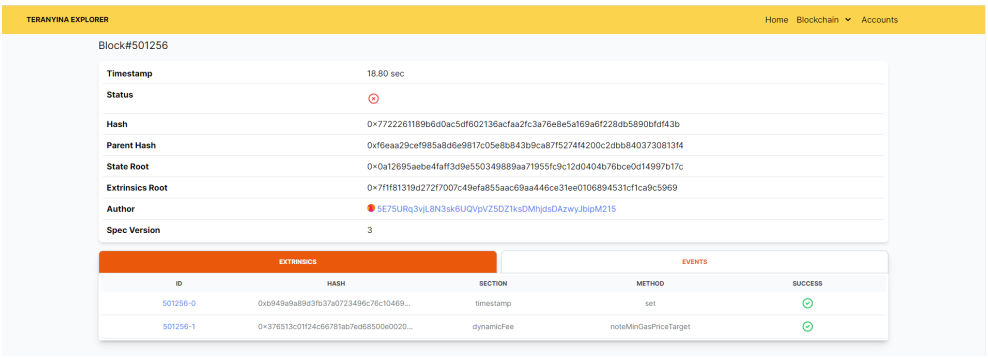

Figura 8.15: Finestra Detall blocs

Igual que les llistes les pàgines de detalls seran similars en estructura, després cada una d'elles mostrarà una cosa diferent.

- Un títol identificant el detall (número de bloc, hash de la transferència...)
- Un seguit de camps.
- Alguns detalls mantindran llistes d'altres col·leccions, com en el cas dels blocs que tindrà una llista d'Extrinsics i una d'Events.

CAPÍTOL 9

Implementació i proves

Una vegada explicat extensivament el disseny de les diferents parts que formen el projecte, en aquest capítol procedirem a explicar com hem implementat el disseny, quins errors ens han sorgit i quines proves hem dut a terme per comprovar la funcionalitat de cada peça que forma el conjunt del projecte.

De la mateixa manera que hem tractat el disseny, també dividirem les diferents implementacions en tres parts: Servidor, Sincronitzador de dades i client.

9.1 Implementació Servidor

Com hem comentat en el capítol 7, el servidor ha estat construït sobre el framework Nest, un framework de Node.js. La nostra base de dades és MongoDB i l'API ha estat construïda amb GraphQL.

Quan inicialitzem o creem un projecte amb Nest.js obtenim la següent estructura:

```
src
    app.controller.ts
    app.module.ts
    app.service.ts
    main.ts
```

Dels darrers 4 elements, només ens interessen el **app.module.ts** i el **main.ts**, ja que els altres dos són exemples de com ha de ser un service i com ha de ser un controller.

- **main.ts**: El fitxer d'entrada de l'aplicació que utilitza la funció principal NestFactory per crear una instància d'aplicació Nest. El main.ts inclou una funció asíncrona, que arrencarà la nostra aplicació.

```
1 import { NestFactory } from '@nestjs/core';
2 import { AppModule } from './app.module';
3
4 async function bootstrap() {
5   const app = await NestFactory.create(AppModule);
```

```
  await app.listen(5000);
}
bootstrap();
```

- **app.module.ts**: Un mòdul és una classe anotada amb un decorador @Module(). El decorador @Module() proporciona metadades que Nest fa servir per organitzar l'estructura de l'aplicació. Cada aplicació té almenys un mòdul, el mòdul arrel. El mòdul arrel és el punt de partida que Nest utilitza per construir el gràfic de l'aplicació, l'estructura de dades interna que Nest utilitza per resoldre les relacions i dependències del mòdul i del proveïdor.

  En el nostre mòdul arrel hi mantindrem els diferents mòduls de GraphQL i MongoDB, els quals ens permetran utilitzar GraphQL i connectar-nos a la base de dades MongoDB respectivament. Aquí també hi guardarem les referències als diferents mòduls que hem creat per organitzar els nostres esquemes. A continuació, veiem el mòdul arrel complet:

```
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';
import { ApolloDriver, ApolloDriverConfig } from
 ↪  '@nestjs/apollo';
import { MongooseModule } from '@nestjs/mongoose';

import { BlockModule } from './block/block.module';
import { ExtrinsicModule } from
 ↪  './extrinsic/extrinsic.module';
import { EventModule } from './event/event.module';
import { TransferModule } from
 ↪  './transfer/transfer.module';
import { AccountModule } from
 ↪  './account/account.module';
import config from "../config/config";

@Module({
  imports: [
    GraphQLModule.forRoot<ApolloDriverConfig>({
      driver: ApolloDriver,
      autoSchemaFile: 'schema.gql',
    }),
    MongooseModule.forRoot(config.mongoDBConstring),
    BlockModule,
```

```
21      ExtrinsicModule,
22      EventModule,
23      TransferModule,
24      AccountModule
25    ],
26  })
27  export class AppModule {}
```

Com hem estat comentant, cada un dels nostres esquemes de la base de dades serà definit per un mòdul importat al mòdul arrel. Aquests mòduls s'encarreguen de mantenir tota l'estructura dels nostres esquemes i només amb un import, el nostre mòdul arrel ja sabrà interpretar tots els esquemes amb els seus resolvers corresponents. Exemple estructura block:

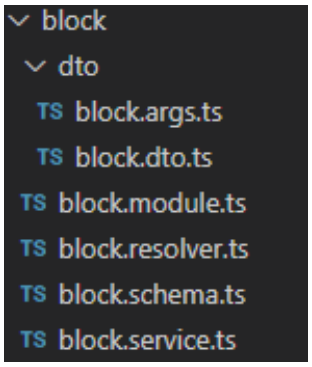

Figura 9.1: Estructura de fitxers block a Nest

- **dtos**: Els dtos ens creen els diferents tipus que fa servir graphQL per interpretar els camps dels esquemes de mongoDB.

```
1  import { ObjectType, Field, InputType, Int } from
   ↪  "@nestjs/graphql";
2  import { type } from 'os';
3  import { Extrinsic } from
   ↪  "src/extrinsic/extrinsic.schema";
4  import { Event } from "src/event/event.schema";
5  import mongoose from "mongoose";
6  import { ExtrinsicType } from
   ↪  "../../extrinsic/dto/extrinsic.dto";
7  import { EventType } from "../../event/dto/event.dto";
8
9
10
```

```
@ObjectType('BlockType')
@InputType('BlockInputType')
export class BlockType {

    @Field(() => Int)
    blockNum: number;

    @Field(() => Number)
    blockTimestamp: number;

    .
    .
    .
}
```

- **module**: El mòdul ens ajuda a organitzar l'aplicació. En el cas del block, té importats tots els mòduls de les classes incrustades (Extrinsic i Event), té la referència a la col·lecció corresponent de la base de dades de mongo i manté una referència a la seva part de l'API de GraphQL (resolver i service).

```
import { Module } from '@nestjs/common';
import { BlockResolver } from './block.resolver';
import { BlockService } from './block.service';

import { ExtrinsicModule } from
 ↪  '../extrinsic/extrinsic.module';
import { EventModule } from '../event/event.module';
import { MongooseModule } from '@nestjs/mongoose';
import { Block, BlockSchema } from './block.schema';

@Module({
  providers: [BlockResolver, BlockService],
  imports: [ExtrinsicModule,EventModule,
   ↪  MongooseModule.forFeature([{ name: Block.name,
   ↪  schema: BlockSchema }])]
})
export class BlockModule {}
```

- **resolver**: Els resolvers proporcionen les instruccions per convertir una operació de GraphQL (una consulta, mutació o subscripció) en dades. Els resolvers saben comunicar-se amb GrapQL, per tant, es guardaran tipus

de GraphQL.

```
import { Args, Int, Mutation, Query, Resolver,
 ↪  Subscription } from '@nestjs/graphql';
import { BlockService } from './block.service';
import { BlockArgs } from './dto/block.args';
import { BlockType } from './dto/block.dto';

@Resolver()
export class BlockResolver {
    constructor(private readonly blockService:
     ↪  BlockService) {}

    @Query(returns => [BlockType])
    async blocks(@Args() blocksArgs: BlockArgs) {
        return this.blockService.findAll(blocksArgs);
    }

    @Query(returns => BlockType, { nullable: true })
    async block(@Args('blockNum') blockNum: Number) {
        return this.blockService.findOne(blockNum);
    }
    @Query(returns => Int)
    async blocksCount() {
        return this.blockService.count();
    }

}
```

- **schema**: Cada esquema s'assigna a una col·lecció MongoDB i defineix la forma dels documents d'aquesta col·lecció.

```
import { Field, InputType, Int, ObjectType } from
 ↪  '@nestjs/graphql';
import { Prop, Schema, SchemaFactory } from
 ↪  '@nestjs/mongoose';
import { Document } from 'mongoose';
import { type } from 'os';
import * as mongoose from 'mongoose';

import { Extrinsic } from
 ↪  '../extrinsic/extrinsic.schema';
```

```
8  import { Event } from '../event/event.schema';
9  import { ExtrinsicType } from
   ↪  '../extrinsic/dto/extrinsic.dto';
10
11 import { EventType } from '../event/dto/event.dto';
12
13 @Schema()
14 @ObjectType()
15 export class Block {
16
17     @Field(()=> Int)
18     @Prop({index: true})
19     blockNum: number;
20
21     @Field(()=> Number)
22     @Prop()
23     blockTimestamp: number;
24
25     .
26     .
27     .
28 }
```

- **service**: Els serveis s'encarreguen de l'emmagatzematge i recuperació de dades a MongoDB. Els serveis saben parlar amb MongoDB, per tant, es guarden esquemes de MongoDB.

  En el servei mantenim tota la lògica de l'API. Utilitzem diferents funcions que deixa al nostre abast mongoose, com find (busca un element que compleixi un filtre. Si no hi ha filtre, retorna tots els elements), skip i limit (retornen X elements de la col·lecció des d'un punt inicial) o populate (completa les dades incrustades amb els camps corresponents)

```
1 import { Model } from 'mongoose';
2 import { Injectable } from '@nestjs/common';
3 import { InjectModel } from '@nestjs/mongoose';
4 import { Block, BlockDocument } from './block.schema';
5 import { BlockArgs } from './dto/block.args';
6 import { BlockType } from './dto/block.dto';
7 import { BlockModule } from './block.module';
8
9
```

```
@Injectable()
export class BlockService {
    constructor(@InjectModel(Block.name) private
     ↪ blockModel: Model<BlockDocument>) {}

    public async findAll(blockArgs: BlockArgs):
     ↪ Promise<Block[]>
    {
        return this.blockModel.find().sort({blockNum:
         ↪ -1}).skip(blockArgs.skip).limit(blockArgs.take).exec();
    }

    public async findOne(num: Number): Promise<Block>
    {
        return this.blockModel.findOne({blockNum: num})
                    .populate('extrinsics')
                    .populate('events').exec();
    }


    public async count(): Promise<Number>
    {
        return this.blockModel.count().exec();
    }

}
```

9.1.1 Proves Servidor

Per poder fer les proves al servidor hem hagut de crear i provar el sincronitzador de dades per poder tenir dades a la base de dades amb què provar l'API.

Cal esmentar, que les proves del funcionament a l'API han sigut possibles gràcies a un playground que ens proporciona el mateix GraphQL.

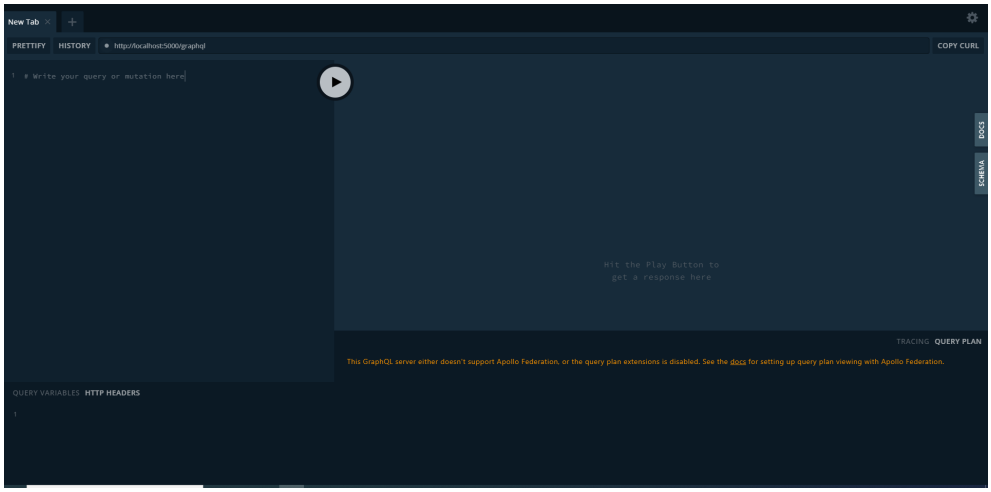

Figura 9.2: Playground GraphQL

Seguidament, mostrarem algunes de les proves que es van fer amb dades ja incorporades a la base de dades.

- Obtenció dels blocs paginats

```
{
  blocks(skip:10, take:5){
    blockNum
  }
}
```

Agafa 5 blocs després de saltar-se 10 blocs. El resultat és el següent:

```
{
  "data": {
    "blocks": [
      {
        "blockNum": 501302
      },
      {
        "blockNum": 501301
      },
      {
        "blockNum": 501300
      },
      {
        "blockNum": 501299
```

```
        },
        {
          "blockNum": 501298
        }
      ]
    }
}
```

- Obtenció d'un sol bloc

```
{
  block(blockNum: 501298){
            blockTimestamp
            finalized
            blockHash
            parentHash
            stateRoot
            extrinsicsRoot
            blockAuthor
            specVersion
            extrinsics{
              blockNum
              extrinsicIndex
              section
              method
              success
              extrinsicHash
            }
            events{
              blockNum
              eventIndex
              extrinsicIndex


              method
            }
    }
}
```

Obtenim la informació del bloc 501298 .Resultat:

```
"data": {
    "block": {
```

```
      "blockTimestamp": 1653856428006,
      "finalized": true,
      "blockHash": "0xab6b4290e6c6d4160842c4858
      820bcdff76bb6a5b3feff8d047d1d0f1aeedc3a",
      "parentHash": "0x4ea054bae7983058f2a952e
      bcd511ed8567eaabae7fc0c43a2821a807ac1ff6d",
      "stateRoot": "0xf529aaeeebc8a31d9b8d
      80d6178f3a1a4450f9adee2e84827353f2cdc4d96357",
      "extrinsicsRoot": "0xd4fc894c1db7db
↪ 938098c04a37d0aa2a1802f90cb47445f59a1410d87affd290",
      "blockAuthor":
      ↪ "5E75URq3vjL8N3sk6UQVpVZ5DZ1ksDMhjdsDAzwyJbipM215",
      "specVersion": 3,
      "extrinsics": [
        {
          "blockNum": 501298,
          "extrinsicIndex": 0,
          "section": "timestamp",
          "method": "set",
          "success": true,
          "extrinsicHash":
          ↪ "0x6f45526d8ef89354d81b74da117
          a0376b8f4ff2b3ec03086f4b83b283c74c8ab"
        },
        {
          "blockNum": 501298,
          "extrinsicIndex": 1,
          "section": "dynamicFee",
          "method": "noteMinGasPriceTarget",
          "success": true,
          "extrinsicHash":
          ↪ "0x376513c01f24c66781ab7ed68500
          e002058424804bfdb31de5592f79a3e2dc03"
        }
      ],
      "events": [
        {
          "blockNum": 501298,
          "eventIndex": 0,
          "extrinsicIndex": 0,
```

```
            "section": "system",
            "method": "ExtrinsicSuccess"
          },
          {
            "blockNum": 501298,
            "eventIndex": 1,
            "extrinsicIndex": 1,
            "section": "system",
            "method": "ExtrinsicSuccess"
          }
        ]
      }
    }
```


- Obtenció de les transaccions d'una account.

```
accountTransfers(accountId:
 ↪  "5E75URq3vjL8N3sk6UQVpVZ5DZ1ksDMhjdsDAzwyJbipM215"){
                blockNum
                hash
                source
                destination
                amount
                success

              }
```

Obtenim les transaccions de l'account 5E75URq3vjL8N3sk6UQVp.... Resultat:


```
{
  "data": {
    "accountTransfers": [
      {
        "blockNum": 12392,
        "hash": "0x7c777cafb3118753d423195242536d23c
        4e5534ef8e82fde94bdf6f1c4b22a0b",
        "source": "",
        "destination": "5E75URq3vjL8N3sk6UQVpVZ5DZ1
        ksDMhjdsDAzwyJbipM215",
        "amount": 1000000000,
        "success": true
```

```
13          }
14        ]
15      }
16  }
```

No ens han sorgit problemes a l'hora de programar el servidor més que saber com funcionaven GraphQL i mongoDB mitjançant prova i error.

Sí que cal comentar, però, que GraphQL no sap tornar valors nulls a no ser que l'hi indiquis. En els casos que pot ser que algun camp sigui null l'hi hem indicat.

9.2 Implementació Sincronitzador de dades

El sincronitzador de dades, bàsicament, serà un script amb un seguit de helpers que anirà incorporant a totes les col·leccions de la base de dades les dades filtrades que ens provenen de la pròpia blockchain Teranyina.

Segons el disseny que hem realitzat al capítol anterior tindrem la següent estructura de fitxers.

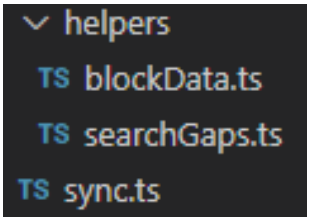

Figura 9.3: Estructura de fitxers del sincronitzador de dades

L'Script sync.ts arrancarà automàticament la funció Run(), la qual seqüencialment executarà les següents funcionalitats:

- Connexió al node de la blockchain i creem una instància de l'API polkadot.js.

```
1  //Connect to node
2      const wsProvider = new
        ↪  WsProvider(config.wsProviderUrl);
3      const api = await ApiPromise.create({ provider:
        ↪  wsProvider });
```

- Connexió a MongoDB.

```
//Db connection
    const db = await
     ↪  mongoose.connect(config.mongoDBConstring);
    mongoose.connection
        .once("open", () => console.log("Connected to
         ↪  Database"))
        .on("error", error => {
            console.log("Couldn't connect to MongoDb
             ↪  Database",error);
        }
    );
```

- Paral·lelització dels processos principals. Es llançaran en paral·lel les següents funcions:

```
//Async main functionalities.
    Promise.all([ findGaps(db,api),
     ↪  processAllAccounts(db,api),
     ↪  listenBlocks(db,api)])
```

Procedim a explicar cadascun dels processos independents que es llancen en paral·lel.

- **findGaps**

```
async function findGaps(db,api: ApiPromise){


  console.log("Gaps Thread Started")

    const lastHeader =  await
     ↪  api.rpc.chain.getHeader();
    console.log(lastHeader.number.toNumber());
    const gaps = await
     ↪  searchGaps(lastHeader.number.toNumber(),db,BlockSchema);
    console.log(gaps);
    for(const gap of gaps){
      let block = gap.l;
      while(block<=gap.r){
        await addBlocksDb(db,api,block,false);
        block++;
      }
```

```
16          console.log("Gaps Thread Started:Gap afegit")
17      }
18
19  }
```

La funció findGaps té com a objectiu retornar i afegir tots aquells rangs de blocs que no han estat afegits a la base de dades, sigui perquè hem iniciat el procés per primer cop o hem parat el procés quan encara no havíem acabat d'inserir tots els blocs i hem parat d'escoltar els nous blocs incorporats.

La funció es comporta de la següent manera: primer busca els rangs de blocs i, una vegada obtinguts, procedeix a inserir-los linealment.

Anteriorment, també paral·lelitzàvem la inserció dels rangs (gaps), però vam tenir problemes quan teníem massa rangs i la memòria de Node arribava al seu límit. Al final es va decidir executar la inserció linealment.

A més a més, abans tampoc fèiem una cerca dicotòmica. Utilitzàvem un aggregate molt complex de mongoDB, el qual havíem programat a mà, que només ens anava ràpid la primera vegada que engegàvem el procés. Després, com més blocs teníem ja inserits a la base de dades, més tardava a trobar els rangs de blocs. Així doncs, es va optar per implementar una cerca dicotòmica, la qual ens triga molt pocs segons a trobar tots els rangs, hi hagi molts o pocs blocs ja inserits.

El cor de la funció findGaps és la cerca dicotòmica que ens troba els diferents rangs de blocs: searchGaps.

- **processAllAccounts**

```
1  async function processAllAccounts(db,api) {
2
3
4    let limit = 50;
5    let last_key = "";
6    let query = await
      ↪  api.query.system.account.entriesPaged({ args: [],
      ↪  pageSize: limit, startKey: last_key });
7    //console.log(query[limit-1]);
8    while(query.length != 0){
9      for(const account of query){
10       let account_id =
          ↪  encodeAddress(account[0].slice(-32));
```

```
11          await
             ↪ addOrReplaceAccount(api,account_id,db,false);
12          last_key = account[0];
13        }
14        query = await
           ↪ api.query.system.account.entriesPaged({ args:
           ↪ [], pageSize: limit, startKey: last_key });
15      }
16      console.log("Accounts afegides");
17    }
```

La funció processAllAccounts té com a objectiu inserir totes les accounts que guarda el sistema de la blockchain.

La funció es comporta de la següent manera: primer obté de forma paginada de 50 en 50 les accounts que tenim registrades a la blockchain, i despreés les processa, afegint-les si no existeixen a la base de dades. Si existeixen, no fa res.

La crida de polkadot.js per extreure les accounts necessita anar paginada, ja que si intentem obtenir totes les accounts de cop, col·lapsaríem el procés. Hem de tenir en compte, que si hi ha moltes accounts en el sistema arribaríem a la màxima memòria Node.js. Havíem tingut el problema de provar amb milions d'accounts i ens retornava una array tan immensa que el mateix Node no la podia processar. En canvi, si anem obtenint les accounts de 50 en 50, la crida és molt ràpida i podem anar inserint accounts d'una manera més eficient.

- **listenBlocks**

```
1   async function listenBlocks(db,api:ApiPromise){
2
3
4     const chain = await api.rpc.system.chain();
5     await api.rpc.chain.subscribeNewHeads(async
       ↪ (lastHeader) => {
6       console.log(`${chain}: last block
         ↪ #${lastHeader.number} has hash
         ↪ ${lastHeader.hash}`);
7
8       //process new Block
9       await
         ↪ addBlocksDb(db,api,lastHeader.number.toNumber(),true);
```

```
10      //Mirar finalitzats
11      const finalizedHash = await
         ↪  api.rpc.chain.getFinalizedHead();
12      await processFinalized(api,db,finalizedHash);
13    });
14  }
```

La funció listenBlocks té com a objectiu anar escoltant els nous blocs incorporats a la cadena i anar-los inserint a la base de dades. A més a més, va actualitzant els blocs que ja han estat finalitzats (blocs firmats per un autor).

La crida api.rpc.chain.subscribeNewHeads escolta permenentment a la cadena per nous blocs.

Per inserir els blocs i tots els seus components tenim una funció que ens obté les dades necessàries d'un bloc concret, a través de Polkadot.js, i ens insereix aquest bloc. No té més misteri. Veiem, per exemple, la funció de inserir un bloc.

```
1  async function addBlocksDb(db,api:
    ↪  ApiPromise,blockNum,updateAccount:boolean) {
2      //db.model('blocks', mongoose.Schema.)
3      const Blockmodel =  db.model('blocks', BlockSchema);
4      const block = new BlockType;
5
6      //Get Block Hash
7      const blockHash = await
        ↪  api.rpc.chain.getBlockHash(blockNum);
8      //Get block
9      const CurrentBlock = await
        ↪  api.rpc.chain.getBlock(blockHash);
10     //Extended block
11     const extendedBlock = await
        ↪  api.derive.chain.getBlock(blockHash);
12     //get extended header for author.
13     //const extendedHeader = await
        ↪  api.derive.chain.getHeader(blockHash);
14     const runtime = await
        ↪  api.rpc.state.getRuntimeVersion(blockHash);
```

```
15      const timestamp = blockNum !== 0?
        ↪  parseInt(extendedBlock.block.extrinsics.find(({
        ↪  method: { section, method } }) => section ===
        ↪  'timestamp' && method ===
        ↪  'set',).args[0].toString(),10,) : 0;
16
17
18      block.blockHash = blockHash.toHex();
19      block.blockNum = blockNum;
20      block.parentHash =
        ↪  CurrentBlock.block.header.parentHash.toHex();
21      block.extrinsicsRoot =
        ↪  CurrentBlock.block.header.extrinsicsRoot.toHex();
22      block.stateRoot =
        ↪  CurrentBlock.block.header.stateRoot.toHex();
23      block.blockAuthor = extendedBlock.author?
        ↪  extendedBlock.author.toString():"";
24      //treure counts
25      block.extrinsicsCount =
        ↪  extendedBlock.extrinsics.length;
26      block.eventCount = extendedBlock.events.length;
27      block.specVersion = runtime.specVersion.toNumber();
28      block.finalized = false;
29      block.blockTimestamp = timestamp;
30
31      //Processar contingut block
32      const allevents = extendedBlock.events;
33      let extrinsics = extendedBlock.block.extrinsics;
34      block.extrinsics = [];
35      block.events = [];
36      //Afsegir parametres que falten
37      await
        ↪  processBlockData(api,db,extrinsics,allevents,blockNum,
38      blockHash,block,timestamp,updateAccount);
39      await Blockmodel.updateOne({blockNum:
        ↪  block.blockNum},{$setOnInsert:block},{upsert:
        ↪  true});
40
41      console.log("Block: " + blockNum + " added")
42
43  }
```

9.2.1 Proves sincronitzador de dades

Per comprovar el funcionament de la part del sincronitzador de dades hem anat fent proves unitàries a cadascuna de les parts que componen el sincronitzador.

1. Primerament, vam construir el procés que escolta els nous blocs juntament amb les funcions d'inserir block, extrinsic, events, i transfers. En aquest punt, el més complicat va ser esbrinar quines crides de l'API de polkadot.js necessitàvem per trobar la informació que nosaltres volíem, ja que la documentació era una mica escassa.

   Per exemple, sabem que les transaccions són una classe d'extrinsic, del pallet (section) balance en concret. Però, el pallet balance té diferents tipus de mètodes per tractar diferents tipus de transaccions: forceTransfer, transfer, transferAll, transferKeepAlive.

   Aquests mètodes no guarden la informació de la mateixa manera, per tant, va caldre un estudi intens d'on es guarden les dades que necessitem per cada un d'aquests mètodes. Per exemple, el següent tros de codi ens mostra on pot ser que trobem l'account destí de la transacció:

```
1 if (JSON.parse(extr.params)[0].id) {
2         transfer.destination =
           ↪  JSON.parse(extr.params)[0].id;
3     } else if (JSON.parse(extr.params)[0].address20) {
4         transfer.destination =
           ↪  JSON.parse(extr.params)[0].address20;
5     } else {
6         transfer.destination =
           ↪  JSON.parse(extr.params)[0];
7     }
```

   Prova funcionament listenBlocks:

```
Teranyina: last block #531420 has hash 0xffb7056c3d1f233d8920fd1c
259fa09dac6a3e623a5b0ff20548ed36a054c0c7
Block: 531420 added
```

   Figura 9.4: Prova funcionament listenBlocks

   Primer escoltem un nou bloc i després s'afegeix a la base de dades. El missatge "Block: 531420 added"significa que el bloc s'ha inserit amb èxit.

2. Quan vam tenir el primer punt funcionant, vam procedir a crear la funció de trobar els diferents rangs de blocs. Aquesta funció, com hem comentat,

teníem un aggregate molt extens de mongoDB i al fer proves vam veure que com més blocs teníem inserits a la base de dades, més lent ens anava la crida. Fins i tot, hi havia moments en què no acabava. El final es va optar per fer una cerca dicotòmica, la qual amb les primeres proves ja vam veure que era extremadament més ràpida.

Prova funcionament findGaps:

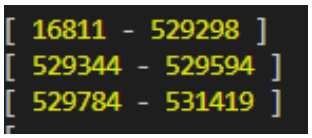

Figura 9.5: Prova funcionament findGaps

El resultat de la cerca dicotòmica ha trobat els anteriors 3 rangs de blocs (gaps).

3. Finalment, vam construir el procés que ens recol·lecta totes les accounts i les afegeix a la base de dades.

Quan ho vam tenir tot acabat, vam procedir a provar-ho tot en conjunt, per veure qüestions de rendiment i eficàcia.

9.3 Implementació Client

El client és la part més independent de tot el projecte. La implementació del client es basa a veure com els altres exploradors organitzen la seva pàgina web i intentar imitar els aspectes comuns entre ells.

El client està construït amb l'ajut de Vite, el qual ens permet estructurar l'aplicació Vue. L'estructura que ens presenta és la següent:

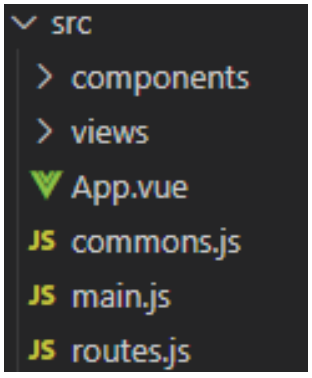

Figura 9.6: Estructura projecte Vite

- **components**: Els components són trossos de codi Vue que ajuden a completar les views, fent el codi més llegible i amb la possibilitat d'utilitzar un mateix component en diferents views. Per exemple, la barra de navegació és un component, ja que és utilitzada per totes les views.
- **views**: Són les pàgines completes, per exemple Home, Blocks, blocks Details... Les view poden estar compostes de diferents components.
- **App.vue**: És la component arrel que s'executa quan inicialitzem el client. Totes les views pengen d'App.vue.
- **commons.js**: Funcions comunes que poden fer servir tots els documents de Vue.
- **main.js**: Normalment, és el fitxer JavaScript que inicialitzarà aquesta component arrel en un element de la vostra pàgina. També és responsable de configurar connectors i components de tercers que es puguin utilitzar a l'aplicació, com és el cas d'Apollo
- **routes.js**: En aquest mòdul de javascript hi emmagatzemem totes les rutes de la nostra pàgina. Cada ruta fa referència a una view en concret.

La implementació de la part client, doncs, tracta d'anar creant views i components que ens ajudin a visualitzar les dades provinents de l'API GraphQL. Pel que fa a la part estètica, ens hem inspirat en els diferents exploradors que podem trobar per la xarxa.

De la implementació, destacarem la part Vue del codi, que és on emmagatzemarem les crides Apollo cap a graphQL i altres funcionalitats de la pàgina.

A continuació, mostrarem un exemple d'aquesta part Vue:

```
export default {
  mixins: [commonjs],
  components: {
    ExtrinsicsTable,
    EventsTable,
    CheckCircleIcon,
    XCircleIcon,
    [Jazzicon.name]: Jazzicon
    },
  data () {
    return {
        // Initialize your apollo data
        transfer: [],
```

```
14          events: [],
15        }
16    },
17    apollo: {
18
19      transfer: {
20        query: gql`query transfer ($hash: String!){
21          transfer(hash: $hash)
22          {
23              blockNum
24              hash
25              source
26              destination
27              amount
28              fee
29              success
30              method
31              events {
32                  blockNum
33                  extrinsicIndex
34                  eventIndex
35                  section
36                  method
37              }
38          }
39        }`,
40        variables () {
41        return {
42          hash: this.$route.params.hash,
43        }
44      },
45      },
46
47    },
48  }
49
50  }
```

Com es pot observar, tenim diferents seccions declarades una darrera de l'altre:

- **mixins**: Els mixins són una manera flexible de distribuir funcionalitats

reutilitzables per als components Vue. En el nostre cas fem referència a aquell mòdul commonjs on guardaven funcions comunes per a totes les pàgines Vue. Aquelles funcions a través de la funcionalitat mixins podran ser cridades en el codi Html.

- **components**: Declaració de tots els components que han ajudat a formar la pàgina o component Vue.

- **data()**: Les dades que tenim a data() són la memòria privada de cada component, on es poden emmagatzemar les variables que es necessitin. En el nostre cas, declarem les dades que ens retornarà Apollo.

- **apollo**: Dins l'objecte apollo es poden declarar consultes que s'enviaran cap al nostre servidor GraphQL quan la pàgina o component Vue sigui cridat pel client. En aquest cas, busca una transferència amb tots els camps indicats.

9.3.1 Proves client

Les proves del client han sigut generalment senzilles, ja que les crides a l'API ja les haurem provat abans al playground de GraphQL. Per tant, només haurem d'enganxar aquestes consultes ja provades a l'objecte Apollo de vue.

Per comprovar que l'objecte que ens arriba és exactament el mateix que rebíem quan fèiem proves al playground de GraphQL, l'ensenyem per pantalla.

{ "__typename": "TransferType", "blockNum": 12392, "hash": "0×7c777cafb3118753d423195242536d23c4e5534ef8e82fde94bdf6f1c4b22a0b", "source": "", "destination": "5E75URq3vjL8N3sk6UQVpVZ5DZ1ksDMhjdsDAzwyJbipM215", "amount": 1000000000, "fee": 125000144, "success": true, "method": "transferKeepAlive", "events": [{ "__typename": "EventType", "blockNum": 12392, "extrinsicIndex": 2, "eventIndex": 2, "section": "system", "method": "NewAccount" }, { "__typename": "EventType", "blockNum": 12392, "extrinsicIndex": 2, "eventIndex": 3, "section": "balances", "method": "Endowed" }, { "__typename": "EventType", "blockNum": 12392, "extrinsicIndex": 2, "eventIndex": 4, "section": "balances", "method": "Transfer" }, { "__typename": "EventType", "blockNum": 12392, "extrinsicIndex": 2, "eventIndex": 5, "section": "system", "method": "ExtrinsicSuccess" }] }

Figura 9.7: Resultat d'haver executat la query transfer

La consulta anterior és el resultat d'haver obtingut una transfer amb un hash concret amb tots els seus elements.

L'únic punt on hem tingut problemes, era saber com obtenir el hash que es trobava a la ruta de la pàgina.

Figura 9.8: Ruta d'una transacció amb hash 0x7c777cafb3118753d423...

Per solucionar-ho només ens cali passar com a variable la seguent variable: this.$route.params.hash.

Pel que fa el cercador, les proves van ser molt senzilles. Només necessitavem saber com identificar una Account, un bloc i una transacció.

- Account: Les accounts a Substrate poden començar per 5 o per 1 seguits de 36 caràcters qualsevols.
- Blocs: Els blocs són números que poden anar des de 0 fins l'infinit.
- Transacció: Una transacció és un hash (seqüència de caràcters començats per 0x).

Vam provar amb diferents entrades el cercador, amb Accounts, transaccions i diferents blocs individualmeent i després, una vegada funcionant les parts individuals és va provar en conjunt.

Lógica del cercador:

```
search() {
        //canviar 1 per 5 quan teranyina
        var accountRegex = new
         ↪  RegExp("(5[0-9a-zA-Z]{47})|(1[0-9a-zA-Z]{46})");
        var blockRegex = new RegExp("\\d+");
        var transferRegex = new
         ↪  RegExp("0x[0-9a-zA-Z]{64}");
        if (transferRegex.test(this.input)){
            this.$router.push('/transfer/'+this.input)
        }
        else if(accountRegex.test(this.input)){
            this.$router.push('/account/'+ this.input)
        }
        else if (blockRegex.test(this.input)) {

            this.$router.push('/block/'+this.input)
        }

        else{
            alert("Invalid input: No match found for
             ↪  Account, Transfer or Block");
        }
    }
```

CAPÍTOL 10

Implantació i resultats

En aquest capítol destacarem els resultats obtinguts vers els objectius inicials del projecte.

La idea base del projecte era crear una interfície web on poguéssim veure d'una manera més amena i dinàmica diferents components de la blockchain Teranyina. Per aquesta raó, tots els resultats seran molt visuals.

Per poder crear el block explorer de la Teranyina es va haver d'estudiar les diferents tecnologies que englobaven la Teranyina: Tecnologia blockchain, Substrate, propòsit de la Teranyina i block explorers.

A continuació, vam desenvolupar la part del servidor i el sincronitzador de dades: models de dades, API, obtenció de les dades del node a través de Polkadot.js.

Finalment, vam crear la interfície web amb totes les seves components. Per fer-ho, ens vam fixar en els altres block explorers que hi ha per Internet.

Actualment, el centre EASY té una web desplegada de la Teranyina on podem trobar diferent informació sobre la mateixa: Els nodes que estat connectats, creació dels faucets, un whitepaper sobre la Teranyina... En aquest entorn serà on s'incorporarà el nostre blockchain explorer.

10.1 Resultats

Primerament, procedirem a comentar els resultats obtinguts a l'hora de programar el servidor i el sincronitzador de dades.

10.1.1 Servidor i Sincronitzador de dades

Pel que fa a la part servidor, no hi ha massa a comentar, ja que és la base sobre la qual està construït el nostre projecte. D'altra banda, el sincronitzador de dades sí que ens proporciona uns resultats interessants.

Quan estàvem desenvolupant el sincronitzador de dades, vam anar deixant diferents marques, en forma de logs, de quan s'engegaven i acabaven els processos, s'afegien blocs a la base de dades...

En la següent figura es mostra el sincronitzador en execució.

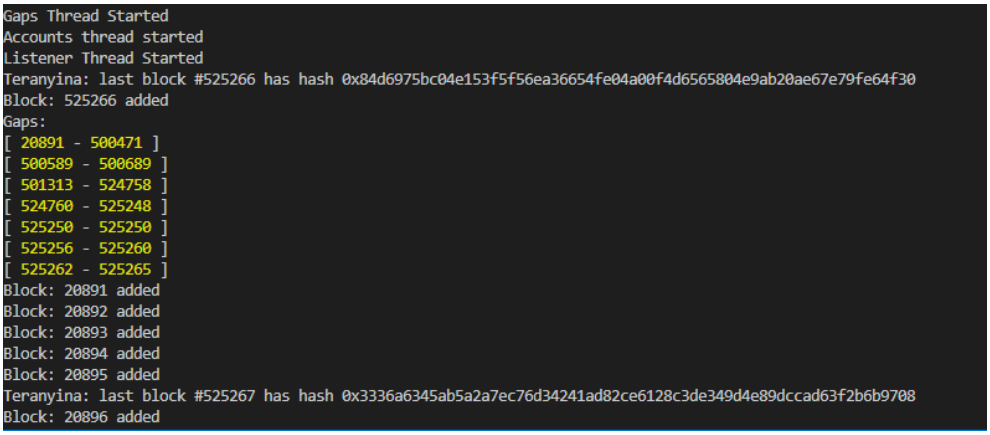

Figura 10.1: Execució del sincronitzador

Com es pot observar, les tres primeres línies ens marquen el començament de cada un dels 3 processos que llancem en paral·lel: findGaps, processAllAccounts i listenBlocks. A continuació, van sortint seqüències de l'execució de cadascun d'aquests processos. Primer, escoltem un bloc que recentment ha incorporat la Teranyina a la seva cadena i l'afegim a la base de dades. Tot seguit, la cerca dicotòmica troba tots els rangs de blocs que falten (part de color groc) i procedeix a inserir-los a la base de dades. Així, repetidament anem escoltant i afegint blocs.

L'execució d'aquest codi és bastant ràpida. Hem calculat el que triga a executar-se l'execució de la Figura 10.1 i ens ha sortit uns 4.5 segons. Per tant, en aquests 4.5 segons hem engegat tots els processos i hem afegit uns 7 blocs.

La següent gràfica ens mostra el temps mitjà que triga a afegir-se un bloc a la base de dades.

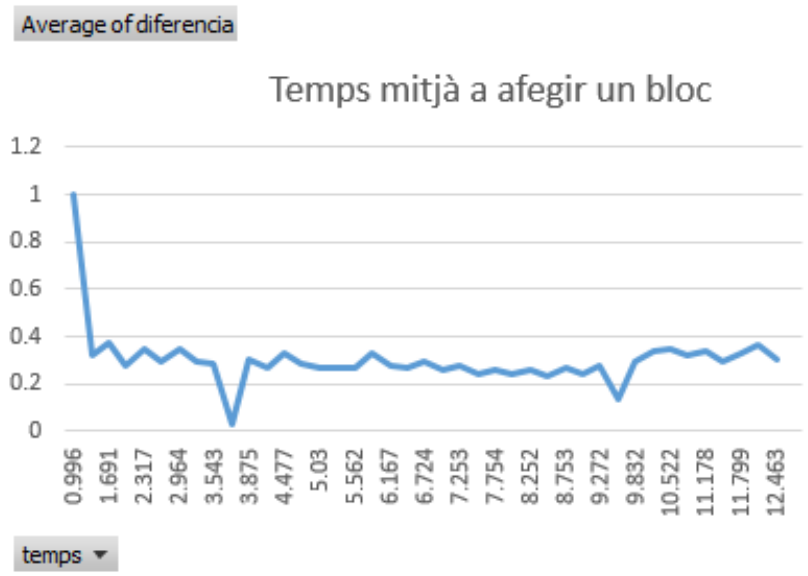

Figura 10.2: Temps mitjà a afegir un bloc a la base de dades

Si ens fixem en la línia de temps (eix X), observem que el primer bloc s'ha afegit al segon 0.996. Això vol dir, que en un segon hem engegat tots els processos, hem trobat tots els rangs (gaps) i s'ha inserit el primer bloc.

Un altre detall interessant a comentar és que el temps que es triga a afegir un bloc segueix una progressió constant. Hem calculat el temps mitjà que triga a afegir-se un bloc i són uns 0.226 segons de mitjana. Si ens fixem en la gràfica, la majoria dels blocs tarden entre 0.4 i 0.2 segons a afegir-se. Fins i tot, hi ha blocs que triguen 0 segons.

10.1.2 Client

A continuació, procedirem a comentar els resultats obtinguts que mostrem a través de la interfície client (web).

Com hem estat comentant anteriorment, per desenvolupar aquesta part client ens hem basat, en diferents exploradors d'Inernet, estèticament i en quines diferents dades mostren per pantalla.

Abans d'entrar en els detalls de la nostra interfície web, però, cal que destaquem varis d'aquests exploradors: etherscan.io, polkascan.io i subscan.

10.1.2.1 Home

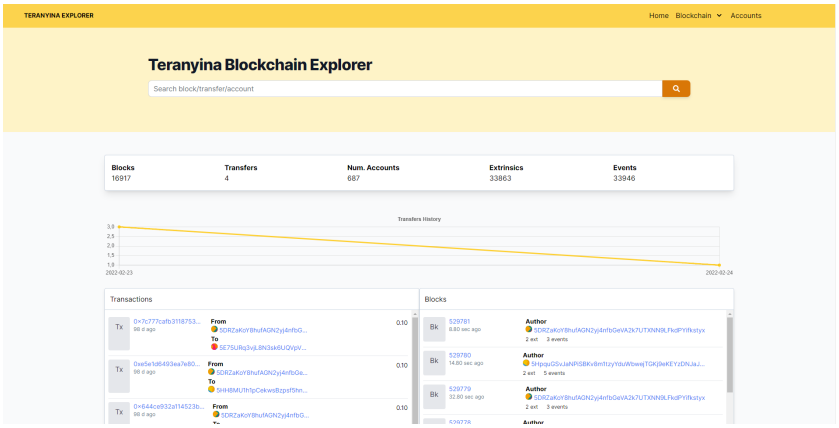

Figura 10.3: Pàgina Home explorador

A primera vista, la nostra interfície web té una pàgina principal on destaquen les següents parts: un cercador, diferents comptadors, una gràfica on es mostra les transaccions històriques, i un seguit de taules que ens mostren les últimes transaccions i els últims blocs afegits a la cadena.

Per desenvolupar aquesta pantalla hem tingut de referent etherscan.io. Creiem que sintetitza molt bé el propòsit de la interfície on apareixen els elements més rellevants.

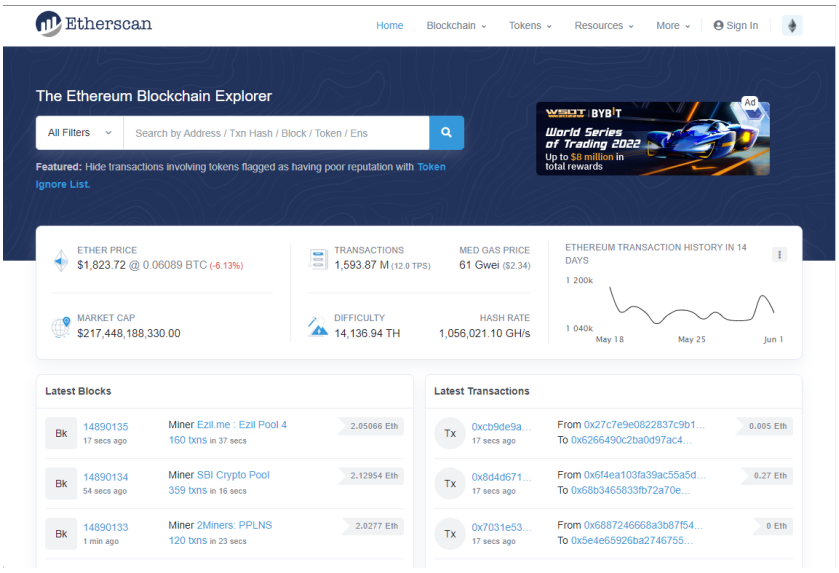

Figura 10.4: Pàgina Home d'Etherescan

El cercador és un cercador simple on hi pots introduir un número de bloc, un hash d'una transacció o un Id d'account. Si el troba, et redirigeix cap al

detall corresponent. Si l'input entrat no correspon a cap hash número de bloc o Id d'account t'indica amb un missatge que l'input no és vàlid. I finalment, si no troba l'element corresponent, et redirigeix a una simple pàgina "Not Found" com la que s'observa a continuació.

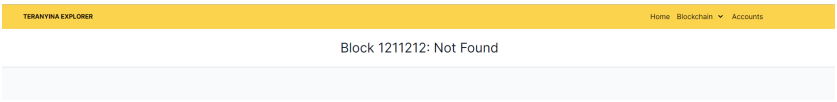

Figura 10.5: Pàgina Not Found

Seguidament, mostrem diferents comptadors dels elements principals de la blockchain: número de blocks totals, transfers, Accounts, extrinsics i events.

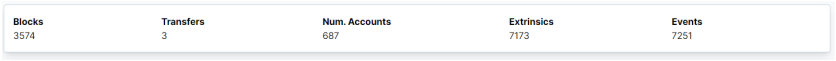

Figura 10.6: Contadors

En aquest apartat es podria millorar el que es mostra, ja que hi ha exploradors que també mostren altres coses. Aquí no ens podem fixar en etherscan, perquè la blockchain d'Ethereum guarda informació que la nostra blockchain de Substrate no té. Doncs, ens vam fixar en exploradors que tinguessin la base de Substrate, com Subscan. Cadascun mostrava diferents elements, però tots acabaven mostrant comptadors de les diferents dades que mantenien, així que finalment es va decidir de moment mostrar només comptadors.

Tot seguit, es mostra un gràfic del nombre de transaccions històriques per dia.

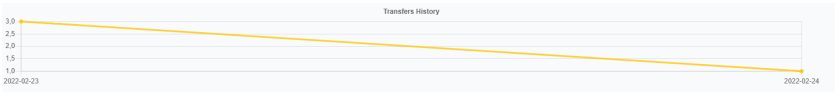

Figura 10.7: Gràfica transaccions històriques

Altres exploradors agafen les transaccions en un període de 30 dies, però nosaltres en tenir molt poques transaccions de moment, vam decidir mostrar un gràfic amb tot l'historial de transaccions de la blockchain.

A la secció on tenim les llistes de les últimes transaccions i els últims blocs, només mostrem la informació més rellevant. Així, l'usuari amb una sola ullada pot veure de què està compost aquell bloc o transacció.

Per exemple, donant un cop d'ull a l'última transacció, primerament, es pot observar el Hash de la transacció i quan fa d'ençà que es va fer. Seguidament,

mostrem el remitent i el receptor de la transacció. I finalment, es mostra la quantitat transferida.

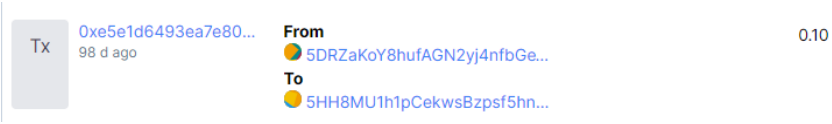

Figura 10.8: informació general d'una transacció

En el cas dels blocs ens passa el mateix: mostrem el número de bloc i el temps transcorregut des que es va afegir a la cadena, l'autor i el número d'extrinsics i events que conté.

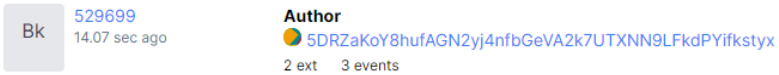

Figura 10.9: informació general d'un block

Cadascuna de les parts en blau ens redirigeix al detall corresponent, sigui el bloc, la transacció o una account.

10.1.2.2 Llistes

Les llistes són les views que mostren un llistat paginat de 25 en 25 d'una col·lecció individual. Per cada un dels ítems de la llista mostrem informació general del que ens trobarem a posteriori al detall.

Aquesta informació general variarà depenent de la col·lecció. Anem a veure les diferents llistes que tenim (una llista per cada col·lecció de la base de dades):

- Blocks

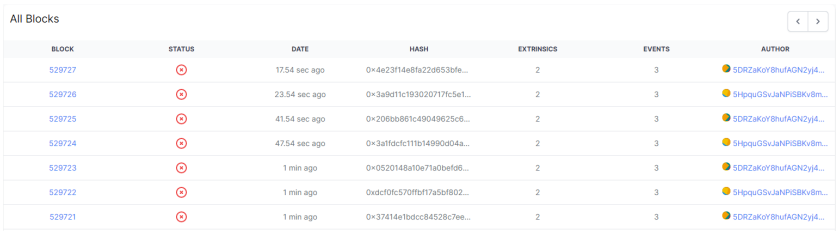

Figura 10.10: Llista de blocs

- Transfers

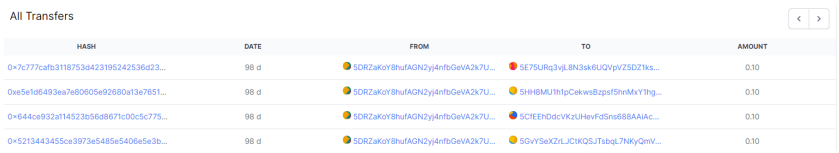

Figura 10.11: llista de transaccions

- Extrinsics

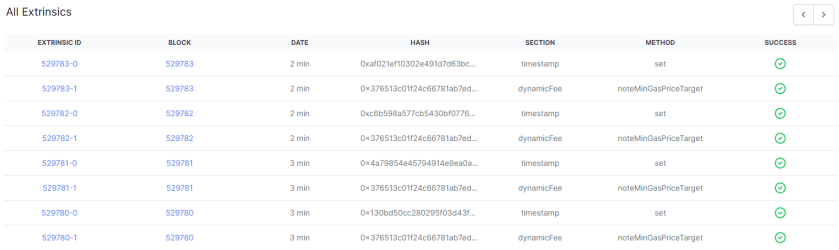

Figura 10.12: llista d'extrinsics

- Events

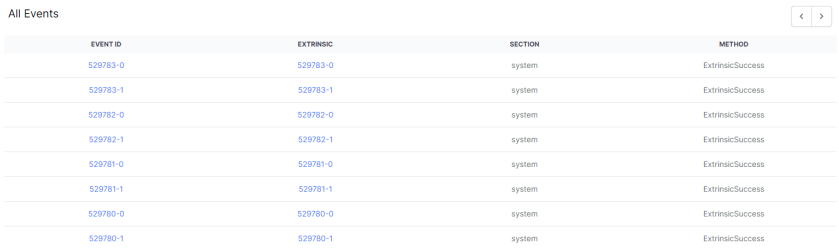

Figura 10.13: llista d'events

- Accounts

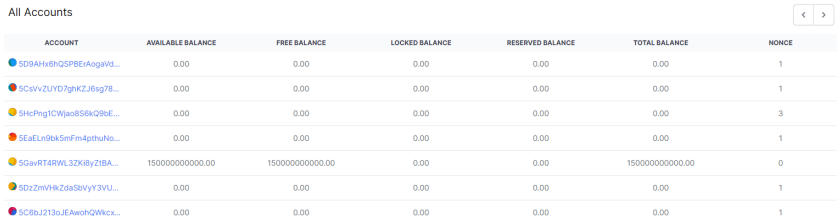

Figura 10.14: llista d'accounts

Per fer les llistes ens hem inspirat en diferents exploradors. En Etherscan i subscan, per l'estètica. I en Subscan, Polkascan i Moonbeam per les dades a mostrar.

A continuació, mostrem les llistes dels blocs a Etherscan i Subscan:

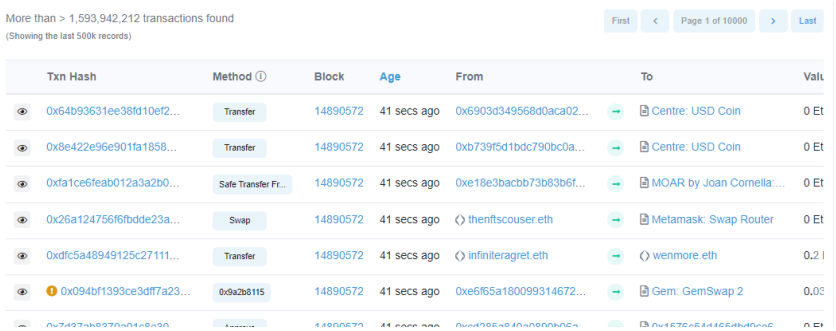

Figura 10.15: Llista Etherescan

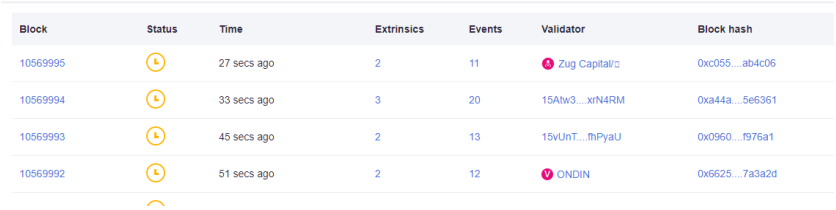

Figura 10.16: Llista Subscan

10.1.2.3 Detalls

Els detalls ens mostren informació detallada d'un document el qual pertany a una col·lecció en particular. Anem a veure els diferents detalls i com es componen.

- Block

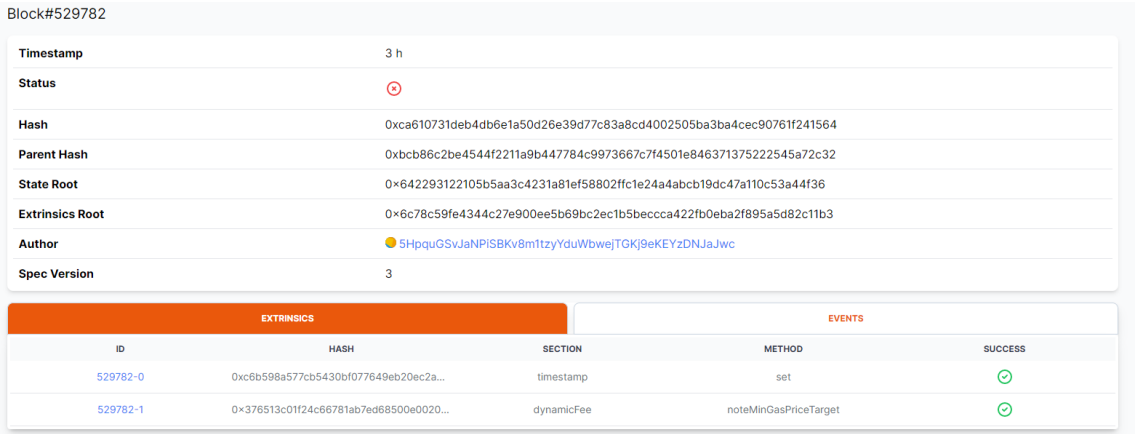

Figura 10.17: Detall block

- Transfer

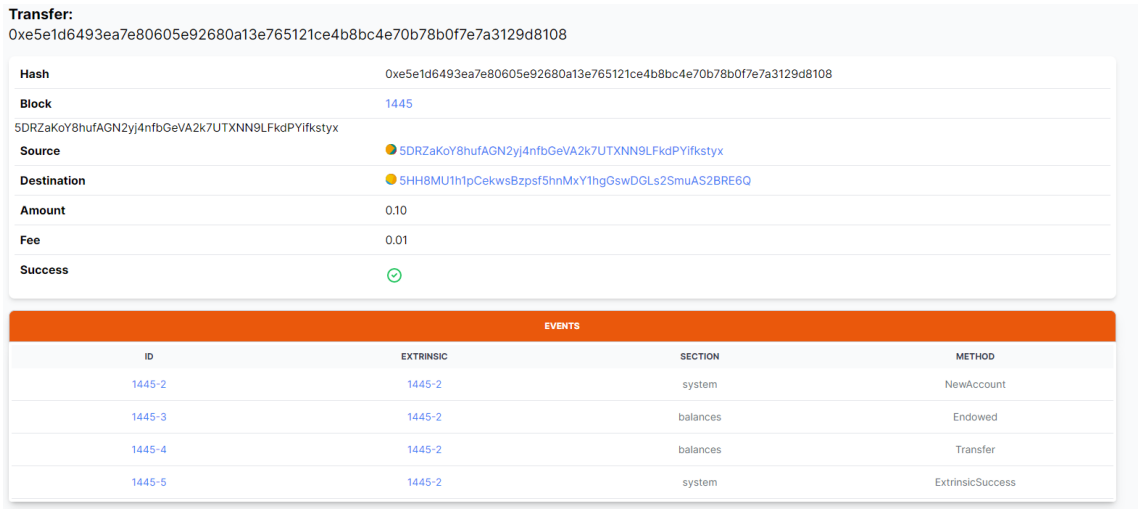

Figura 10.18: Detall transfer

- Extrinsic

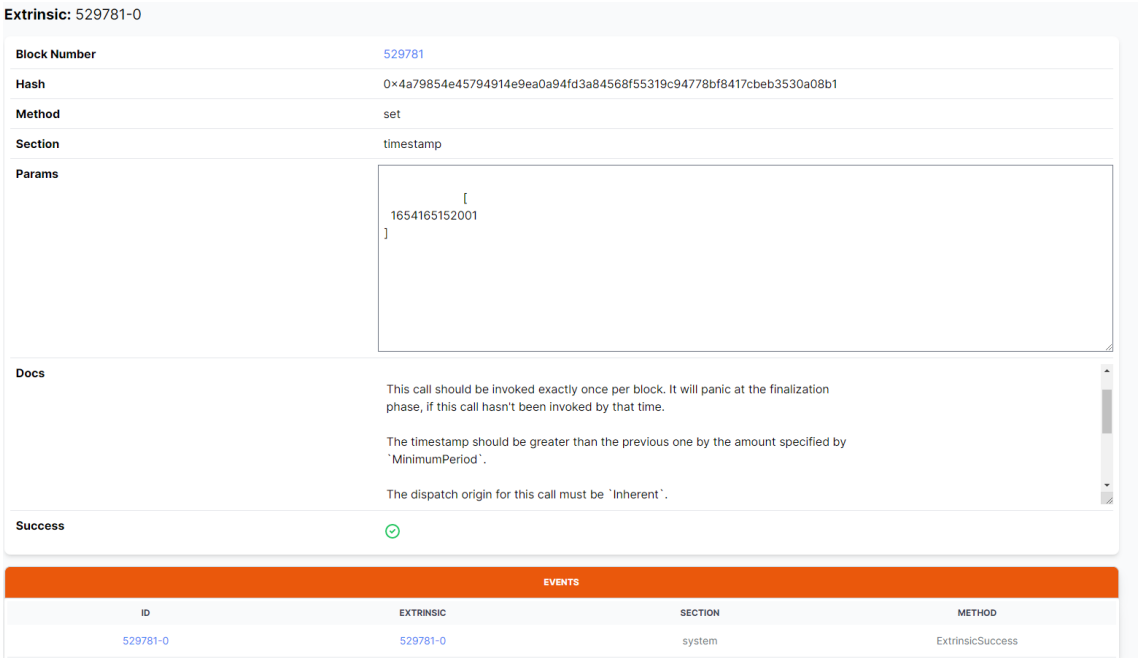

Figura 10.19: Detall extrinsic

- Event

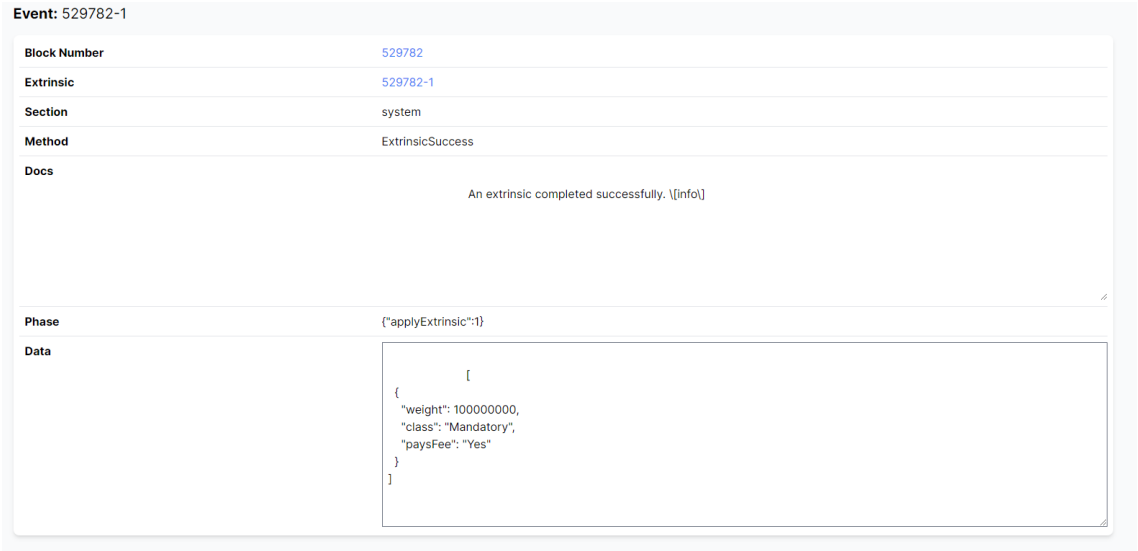

Figura 10.20: Detall Event

- Account

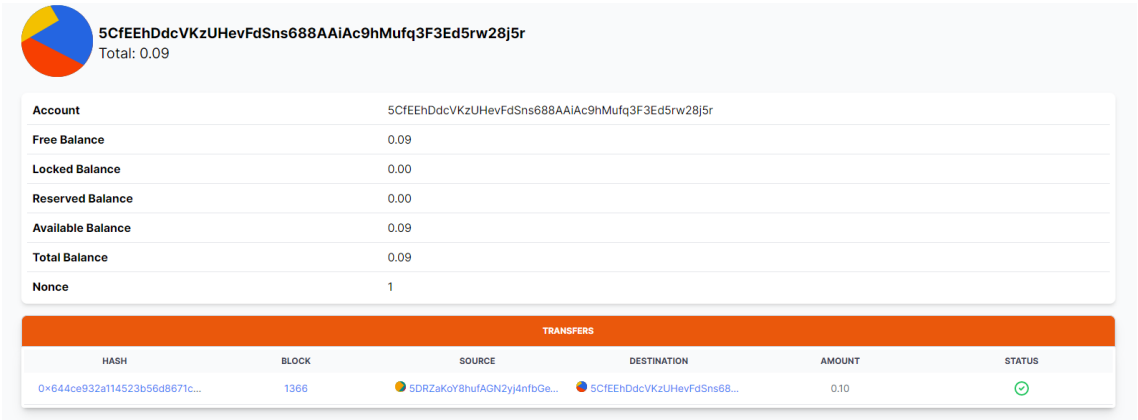

Figura 10.21: Detall account

Com podem observar, els detalls anteriors es comporten tal com havíem definit en el nostre disseny. Els blocs guarden extrinsics i events, les accounts tenen un llistat de les seves transaccions...

Així mateix, igual que amb les llistes, pels detalls també ens hem inspirat en diferents exploradors. En aquest cas, però, només hem agafat com a fons inspiratives els exploradors basats en Substrate, ja que les dades que necessitem mostrar són les mateixes o relativament semblants.

10.1.2.4 Rendiment Client

El rendiment del client és excepcional. Totes les pàgines es carreguen instantàniament, siguin llistes detalls o la pròpia pàgina Home. El cercador també ens troba resultats al moment.

Gràcies a la indexació amb mongoDB i la utilització de GraphQL, les consultes s'executen i responen molt ràpid, la qual cosa implica que no tinguem pràcticament cap càrrega lenta de pàgina.

Per acabar, cal destacar que la totalitat de la interfície web és responsive. Pot ser oberta des de qualsevol dispositiu de qualsevol mida.

CAPÍTOL 11

Conclusions

En general, podem dir que els objectius del projecte s'han complert. S'ha creat un blockchain explorer compost per: una base de dades on s'incorporen dades a través d'un sincronitzador que les va llegint d'un node de la blockchain Teranyina, una API d'accés a aquestes dades i una interfície web que permet consultar i navegar per aquestes dades visualment.

Com hem estat remarcant durant tota la documentació d'aquest treball, la Teranyina encara està en creixement i espera incorporar moltes més funcionalitats en un futur. L'explorador que hem creat, de moment treballa amb les dades que actualment suporta la Teranyina. Si en un futur es volen incoporar noves funcionalitats a la Teranyina, com per exemple NFTs, caldrà una actualització de l'explorador actual.

Sí que és veritat que en el nostre explorador no hem incorporat la part de la Virtual Machine d'Ethereum (EVM), Contractes i transaccions d'Ethereum. Bàsicament, vam creure que tota aquesta part se'ns escapava de l'abast del projecte, ja que implementar la màquina virtual d'Ethereum implicava estudiar com tracta les diferents dades Ethereum, què són aquestes dades i esbrinar com incorporar-les a un model. Possiblement, ens caldria un model de dades nou, com hem fet amb la part natural de Substrate, per això es va descartar del projecte final.

En conclusió, la implementació del nostre explorador ha resultat bastant reeixida, i compleix els objectius amb eficiència: el sincronitzador ens incorpora un bloc cada 0.296 segons de mitjana, la nostra API respon ràpidament a qualsevol consulta i el nostre client també presenta totes les dades a l'instant quan són demanades.

CAPÍTOL 12

Treball futur

En tots els projectes sempre hi ha coses a millorar, sobretot en projectes com aquest, que s'han començat des de zero, sense cap base. Tenint en compte que la Teranyina encara està en procés de desenvolupament, pot ser que en un futur calgui millorar o afegir algun nou aspecte. A més a més, com hem comentat, no s'ha implementat la part del pallet d'Ethereum Virtual Machine (EVM) que guarda tota la informació sobre smart contracts i diferents transaccions d'Ethereum. En un futur se li podria incorporar, possiblement creant un altre model de dades especialment dissenyat per emmagatzemar els camps d'Ethereum.

A continuació llistem alguns dels aspectes millorables o evolucions de la nostra implementació:

- Incorporar la part d'Ethereum virtual machine.
- Filtrar Extrinsics per pallets com fan altres block explorers.
- Millorar el cercador.
- Mostrar diferentes gràfiques, com fem amb el nombre de transacions històriques.
- Incorporar diferentes estadístiques a la pàgina d'inici a més dels contadors.

Bibliografia

[1] 101blockchains. What Is a Block Explorer?, (Accedit: Març 2022). Disponible a https://101blockchains.com/best-blockchain-explorers/. (Not cited.)

[2] Vue Apollo. Vue Apollo: Integrate GraphQL in your Vue.js apps!, (Accedit: Maig 2022). Disponible a https://apollo.vuejs.org/. (Not cited.)

[3] Blockchair. Blockchain explorer, analytics and web services, (Accedit: Març 2022). Disponible a https://blockchair.com/. (Not cited.)

[4] centre EASY. Blockchain Teranyina, (Accedit: Març 2022). Disponible a http://teranyina.centreeasy.com/. (Not cited.)

[5] codecademy. What is Blockchain?, (Accedit: Març 2022). Disponible a https://www.codecademy.com/resources/blog/what-is-blockchain/. (Not cited.)

[6] Cryptopedia. What Is a Block Explorer?, (Accedit: Març 2022). Disponible a https://www.gemini.com/cryptopedia/what-is-a-block-explorer-btc-bch-eth-ltc. (Not cited.)

[7] Etherescan. The Ethereum Blockchain Explorer, (Accedit: Març 2022). Disponible a https://etherscan.io/. (Not cited.)

[8] Simply Explained. How does a blockchain work - Simply Explained, (Accedit: Març 2022). Disponible a https://www.youtube.com/watch?v=SSo_EIwHSd4. (Not cited.)

[9] Express. Express: Fast, unopinionated, minimalist web framework for Node.js, (Accedit: Abril 2022). Disponible a http://expressjs.com/. (Not cited.)

[10] gobitfly. etherchain-light, (Accedit: Abril 2022). Disponible a https://github.com/gobitfly/etherchain-light. (Not cited.)

[11] GraphQL. GrapQL, (Accedit: Abril 2022). Disponible a https://graphql.org/. (Not cited.)

[12] Heroicons. Heroicons: Beautiful hand-crafted SVG icons,by the makers of Tailwind CSS., (Accedit: Maig 2022). Disponible a https://heroicons.com/. (Not cited.)

[13] hyperledger. blockchain-explorer, (Accedit: Abril 2022). Disponible a https://github.com/hyperledger/blockchain-explorer. (Not cited.)

[14] IBM. What is blockchain technology?, (Accedit: Març 2022). Disponible a https://www.ibm.com/topics/what-is-blockchain#:~:text=Blockchain. (Not cited.)

[15] Investopedia. Blockchain Explained, (Accedit: Març 2022). Disponible a https://www.investopedia.com/terms/b/blockchain.asp. (Not cited.)

[16] Medium. What is the Blockchain Block Explorer and why it is so important, (Accedit: Març 2022). Disponible a https://medium.datadriveninvestor.com/what-is-the-blockchain-block-explorer-and-why-it-is-so-important-cd449e5daf07. (Not cited.)

[17] Microsoft. Simple Gantt Chart, (Accedit: Maig 2022). Disponible a https://templates.office.com/en-us/Simple-Gantt-Chart-TM16400962. (Not cited.)

[18] MongoDB. MongoDB, (Accedit: Abril 2022). Disponible a https://www.mongodb.com/. (Not cited.)

[19] mongoose. mongoose: elegant mongodb object modeling for node.js, (Accedit: Abril 2022). Disponible a https://mongoosejs.com/. (Not cited.)

[20] Moonbeam. Moonbeam Chain Explorer, (Accedit: Març 2022). Disponible a https://moonbeam.moonscan.io/. (Not cited.)

[21] Nest.js. Nest.js, (Accedit: Abril 2022). Disponible a https://nestjs.com/. (Not cited.)

[22] nodejs. Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine., (Accedit: Abril 2022). Disponible a https://nodejs.org/en/. (Not cited.)

[23] overleaf. overleaf documentation, (Accedit: Abril 2022). Disponible a https://www.overleaf.com/learn. (Not cited.)

[24] Polkadot. Intro to Substrate - The Modular Framework for Building Custom Blockchains on Polkadot or Kusama, (Accedit: Abril 2022). Disponible a https://www.youtube.com/watch?v=-6BBIr-DmI4. (Not cited.)

[25] Polkadot. Polkadot: Are You Ready to Start Building?, (Accedit: Abril 2022). Disponible a https://www.youtube.com/watch?v=_-k0xkooSlA. (Not cited.)

[26] polkadot.js. polkadot.js, (Accedit: Abril 2022). Disponible a https://polkadot.js.org/. (Not cited.)

[27] polkadot.js. polkadotApiExplorer, (Accedit: Maig 2022). Disponible a https://polkadot.js.org/apps/#/explorer. (Not cited.)

[28] Polkascan. Polkadot explorer, (Accedit: Març 2022). Disponible a https://explorer.polkascan.io/polkadot. (Not cited.)

[29] Vue Router. Vue Router: The official router for Vue.js., (Accedit: Abril 2022). Disponible a https://router.vuejs.org/. (Not cited.)

[30] shawntabrizi. What is Substrate?, (Accedit: Març 2022). Disponible a https://www.shawntabrizi.com/substrate/what-is-substrate/. (Not cited.)

[31] Subscan. Aggregate Substrate ecological network High-precision Web3 explorer, (Accedit: Març 2022). Disponible a https://www.subscan.io/. (Not cited.)

[32] subscan explorer. subscan-essentials, (Accedit: Abril 2022). Disponible a https://github.com/subscan-explorer/subscan-essentials. (Not cited.)

[33] Substrate. Subtrate Home, (Accedit: Març 2022). Disponible a https://substrate.io/. (Not cited.)

[34] TailwindCss. Moonbeam Chain Explorer, (Accedit: Abril 2022). Disponible a https://tailwindcss.com/. (Not cited.)

[35] Parity Technologies. Substrate: Blockchain Framework in Rust , (Accedit: Abril 2022). Disponible a https://www.youtube.com/watch?v=NrG3co6UWEg. (Not cited.)

[36] How to geek. What Is a “Blockchain”?, (Accedit: Març 2022). Disponible a https://www.howtogeek.com/335814/what-is-a-blockchain/. (Not cited.)

[37] Vite. Vite Next Generation Frontend Tooling, (Accedit: Març-Abril 2022). Disponible a https://vitejs.dev/. (Not cited.)

[38] Vue.js. The Progressive JavaScript Framework, (Accedit: Març-Abril 2022). Disponible a https://vuejs.org/. (Not cited.)

[39] WebDollar. blockchain-explorer , (Accedit: Abril 2022). Disponible a https://github.com/WebDollar/blockchain-explorer. (Not cited.)